StormRider: Harnessing "Storm" for Social Networks

Vaibhav Khadilkar, Murat Kantarcioglu,
Bhavani Thuraisingham
The University of Texas at Dallas

Contents

List of Figures

List of Tables

Abstract

The diverse types of social networks that are available today have made the once exclusive area of web publishing into a routine activity for all people. The focus of online social media companies has thereby moved away from "content generation" towards finding effective methodologies for "content storage, retrieval and analysis". An efficient solution to this problem must therefore be able to handle these tasks automatically, and adapt to a network continuously evolving over time. Towards this end, in this report we present StormRider, a framework that uses existing Cloud Computing and Semantic Web technologies to provide application programmers with tools that store, query and analyze large-scale, evolving social networks. In particular, StormRider provides automated support for these tasks, thereby allowing a richer assortment of use cases to be implemented on the underlying evolving social networks.

Chapter 1

Introduction

The World Wide Web (WWW) was designed as a "web" of "hypertext documents" to be viewed by "browsers" using a client-server architecture [1]. The guiding principle behind the WWW was to allow anyone to publish information on any topic. The published information was then accessible to anyone who wished to view it. The advent of online social media has made this principle more relevant than ever before. The explosion in the number of online social media applications has turned the once privileged realm of web authoring and publication into a commonplace activity. Furthermore, this ease of publishing information online has brought new meaning to the terms "content generation" and "content sharing", with vast quantities of data being generated and shared at any given instant in time. Moreover, people can join this online movement by freely contributing whatever information they wish to share, thus enabling the creation of online communities based on diverse criteria such as location, age-group, mutual interests, *etc.*

The various players in the online social media space (*viz.* Facebook, Twitter, Google, *etc.*) no longer concern themselves with the realm of "data collection". Their main challenge has now become finding an effective answer to the following question: How to store the vast quantity of information efficiently and how to effectively make sense of all the information that is being stored online? To a certain extent, the evolution of Cloud Computing and Semantic Web technologies along with the historical field of social network analysis provide us with a unique opportunity to find a practical answer to this question. The emergence of Cloud Computing technologies such as Hadoop, HBase, Storm, *etc.* have made it possible to store, manage and perform effective analyses on large amounts of information. On the other hand, Semantic Web technologies have enabled rich and expressive representations of data that can be automatically read and processed by machines. Finally, the field of social network analysis provides us with a variety of ways with which to systematically study various characteristics of social networks.

There has been a significant amount of research done to develop solutions to the above question [2, 3]. Additionally, several open-source tools have also

been developed to address these issues [4, 5, 6]. However, this entire body of research work views a social network as a *series of snapshots* separated over distinct periods of time. Therefore, the tasks of storage, retrieval and analysis of a network need to be manually performed over each snapshot. In reality, online social networks *continuously evolve* and therefore, their storage, retrieval and analysis should be automatically performed as the network is evolving. A software solution that addresses this problem can now support real-world use cases such as the following:

Example 1: Consider a scenario in which government agencies track potential terrorists by monitoring publicly available online information as found on social networking websites, blogs, discussion groups, *etc.* Further, by using such information agencies are able to form a network of interactions between various people including potential terrorists. Also, let us assume that agencies have already defined a list of potential terrorists by using other sources of intelligence (*e.g.* ground surveillance). In this scenario, it is now important for agencies to continuously monitor the neighborhood of users who have been identified as potential terrorists as the network evolves over time.

Example 2: Consider a scenario in which hospitals want to keep track of the medical history of their patients. This task entails the following sub-tasks: (i) When a patient visits the hospital for the first time, a new medical history file is created for the patient. (ii) For every subsequent patient visit, the history file is updated with newer information such as the type of medical tests conducted, the doctor/nurse administering the test, *etc.* Let us assume that we have a system that records such information for every patient. It should also be evident that this type of information can naturally be represented as a network in which patients, doctors, nurses, *etc.* can be considered as nodes, while relationships between them (*e.g* nurse administers a medical test to a patient) form the links. In this scenario, if we can store and access all versions of the medical history of a patient, we can subsequently move back and forth between these versions to trace for example, the effect of a new drug that was administered to the patient.

In this technical report, we present StormRider[1], a framework that uses Cloud Computing and Semantic Web technologies to store and retrieve data as a network is evolving. In addition, StormRider also provides users with the ability to conduct some fundamental social network analysis, *viz.*, centrality estimation, on the underlying networks. Furthermore, users can store, query and analyze networks of their choice by creating custom-built implementations of the basic interfaces provided in the StormRider framework.

The main focus of our work is the creation of a scalable framework that supports diverse use cases such as the ones presented in Examples 1 and 2 above. Towards this end, we have made use of the Storm[2] framework as one of the basic building blocks of StormRider. Storm plays a key role in allowing StormRider to automatically store, query and analyze data as the underlying network evolves over time. In addition, we make use of the Jena-HBase[3] [7]

[1] https://github.com/vaibhavkhadilkar/stormrider
[2] https://github.com/nathanmarz/storm
[3] https://github.com/vaibhavkhadilkar/hbase-rdf

framework to store network data in a RDF representation as well as to query the data using the SPARQL query language. Finally, we use the idea of views materialized in HBase to allow faster computation of centrality metrics of nodes in a network.

Our contributions: StormRider framework provides the following:

- Allows the storage, query and analysis of large-scale, evolving networks.
- Enables the application of features such as inference, reification, property-path queries, *etc* to evolving networks.
- Performs some fundamental analysis, *viz.*, centrality estimation, on networks.
- Supports query execution for tasks such as comparing network structure at different points of time. (*e.g.*, network structure last year *vs.* this year).
- Provides application programmers with simple interfaces with which they can store, query and analyze networks of their choice.

The rest of this report is organized as follows: Chapter 2 presents an overview of the tools and technologies that we have used in building the StormRider framework. In chapter 3 we present the architectural details of StormRider. This chapter also highlights sample Storm topologies that we have implemented for the Twitter network. These topologies enable automatic storage, retrieval and analysis of the Twitter network as the underlying data changes. Next, a preliminary performance evaluation of our prototype is presented in chapter 4. Then, chapter 5 presents a summary of the work done in the fields of efficient storage, retrieval and analysis of large-scale social networks in the context of existing Cloud Computing and Semantic Web tools. Finally, we present our conclusions and directions for future work in chapter 6.

Chapter 2

Overview of Tools and Technologies Used

2.1 Overview of RDF

The Resource Description Framework (RDF)[1] is a language that is used to represent metadata about data on the World Wide Web. RDF can not only be used to represent metadata about Web resources but also to represent information about the resources themselves. The RDF language defines a simple graph data model to capture relationships between resources using the concept of a triple. A triple consists of a *subject*, *predicate* and an *object* and a set of such RDF triples forms a RDF graph. A RDF triple asserts that the relationship represented by the *predicate* is held between the *subject* and *object* of the triple.

The RDF data model is extremely flexible since a resource is not constrained based on its type or the properties that can be associated with it. Moreover, RDF uses the open-world assumption that allows anyone to make statements about any resource. The RDF data model is different from other data models in the sense that it does not depend on fixed schemas or type hierarchies.

The challenge for the RDF data model is then to provide a scalable and efficient persistent storage scheme. A simple approach could be to convert the RDF data into XML using the work available on optimized storage of XML. However, there are several drawbacks to this approach. The first is that there are several XML serializations available for the same RDF graph which makes retrieval difficult. The second is that this conversion process is unable to support advanced features of RDF such as reification and the ability to treat properties as resources.

The use of relational or object databases for RDF storage and retrieval may also not be suitable in all cases. This is because the semantics of these data models is different from the open and flexible RDF data model. To this end, in this report we use Jena-HBase, a distributed, column-oriented store that uses

[1] http://www.w3.org/RDF/

HBase to provide persistent storage for RDF data. A crucial advantage of HBase over other models is that HBase does not require a fixed definition of data types during schema creation.

2.2 Overview of Jena

The Jena[2] framework provides a rich API for building and querying RDF graphs. The API includes an I/O mechanism for N3, N-TRIPLE and RDF/XML formats and support for the SPARQL query language. Jena also allows exposing RDF data as a SPARQL endpoint over the web using the Joseki SPARQL server. Jena also provides functionality to support the RDFS and OWL languages through the Jena Inference API. Finally, Jena also provides support for reification through the Jena Reification API. A user can choose to store RDF graphs either in memory or in some form of disk-based persistence provided by Jena.

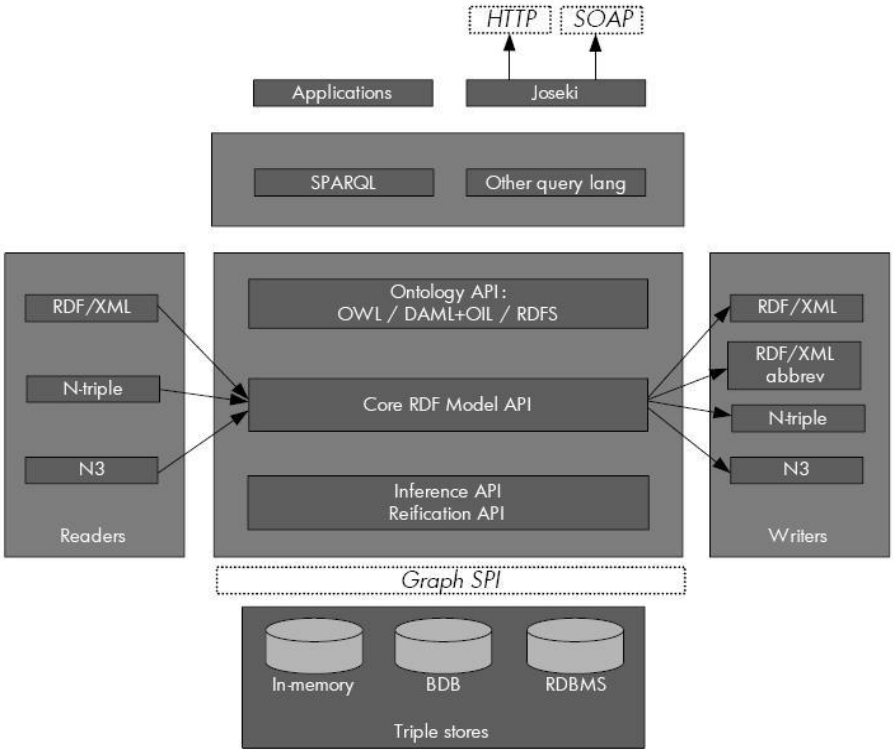

Figure 2.1: Jena Architectural Overview

[2]http://jena.sourceforge.net/

Figure 2.1 provides an overview of the Jena architecture. A user application usually operates with the Core RDF Model API. At this level, a user application only deals with "Statements" in a Model. These statements are like English sentences and therefore, a user is abstracted from the low-level operations on triples of a RDF graph. In this report, we focus on a new implementation, Jena-HBase, for a RDF graph using the HBase distributed, versioned, column-oriented store.

In the Jena framework, any new graph implementation essentially needs to implement the following three operations: (i) An "add" operation that stores a RDF triple in a graph. (ii) A "delete" operation that removes a RDF triple in a graph. (iii) A "find" operation that retrieves all triples that match a given <S, P, O> pattern. Any SPARQL query is converted into a set of "find" operations over the underlying RDF graph. Each of the components S, P or O of <S, P, O> can be a constant, a variable or a wildcard.

Our HBase graph implementation is based on the idea of pluggable storage layouts that was introduced as part of the Jena SDB model architecture. In this approach, each storage layout is implemented as a store and several operations are provided on a store. These include loading-unloading triples and formatting the store among others.

2.3 Overview of Jena-HBase

Jena-HBase[3] provides a HBase-backed storage subsystem for use with the Jena framework [7]. Jena-HBase closely follows the design architecture of Jena SDB by using the concept of a store to provide data loading and querying capabilities on underlying HBase tables. A store represents a single RDF dataset and can be composed of several RDF graphs, each with its own storage layout. A layout uses several HBase tables with different schemas to store RDF triples; each layout provides distinct advantages/disadvantages in terms of query performance/storage. A distinct advantage of using such a pluggable approach is that it is easy for application programmers to plug-in their own storage layouts into Jena-HBase. Additionally, Jena-HBase provides support for other operations such as reification, inference, SPARQL query processing, *etc.*

Figure 2.2 presents an overview of the Jena-HBase architecture. Since Jena-HBase uses the concept of a store, composed of several RDF graphs where each graph can have its own storage layout, all operations on a RDF graph are implicitly converted into operations on the underlying layout. These operations include the following: (a) Formatting a layout (i.e. deleting all triples while preserving the tables in a layout) as represented by the **Formatter** block. (b) Loading-unloading triples into a layout (**Loader** block). (c) Querying a layout for triples that match a given <S, P, O> pattern using the **Query Runner** block. (d) Additional operations include the following: (i) Maintaining a HBase connection (**Connection** block). (ii) Maintaining configuration information for each RDF graph (**Config** block).

[3]https://github.com/castagna/hbase-rdf, https://github.com/vaibhavkhadilkar/hbase-rdf

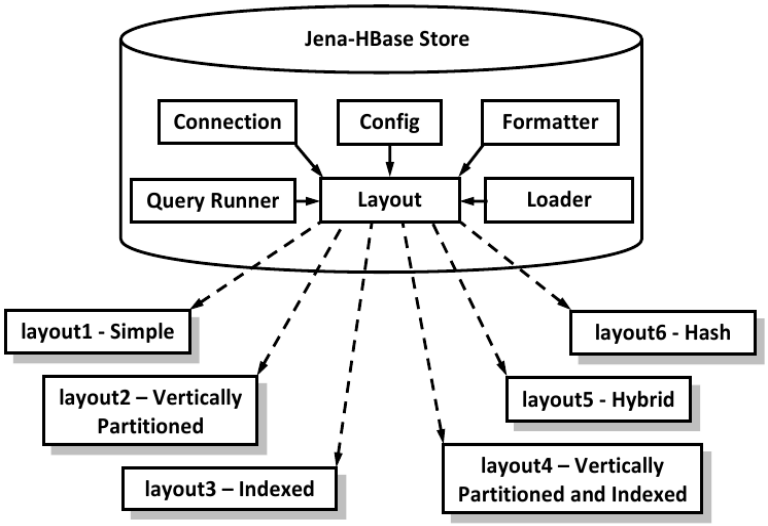

Figure 2.2: Jena-HBase Architectural Overview

We now give a brief summary of the storage schemas used by the various layouts in Table 2.1. A more detailed description of each layout along with ways in which they handle query processing operations is given in [7].

2.4 Overview of Hadoop

Apache Hadoop[4] is a Java-based open source framework that allows storage and processing of large amounts of data. Hadoop uses the Hadoop Distributed File System (HDFS) which is an open source system based on the Google File System [8] as its underlying storage mechanism. HDFS provides several advantages such as data replication and fault tolerance.

HDFS uses a master/slave architecture that consists of a single namenode and several datanodes. The namenode manages the entire file system by keeping track of how blocks of files are distributed over all nodes. A datanode runs on every node and is used as the storage mechanism for that node. Hadoop then uses the MapReduce paradigm to process the data stored in the HDFS in a parallelized fashion using a cluster of nodes.

The MapReduce framework consists of a single JobTracker (on the master) and several TaskTracker's (usually one per slave). The JobTracker is tasked with the responsibility of scheduling a MapReduce job among the slaves while a TaskTracker is responsible for executing the sub-task given to its hosting slave. A MapReduce job consists of three phases: (i) A **Map** phase in which each slave performs some computation on the parts of the input that it has

[4]http://hadoop.apache.org/

Table 2.1: Storage schemas for Jena-HBase layouts

Layout Type	Storage Schema
Simple	Three tables each indexed by subjects, predicates and objects
Vertically Partitioned (VP)	For every unique *predicate*, two tables, each indexed by subjects and objects
Indexed	Six tables representing the six possible combinations of a triple namely, SPO, SOP, PSO, POS, OSP and OPS
VP and Indexed	VP layout with additional tables for SPO, OSP and OS
Hybrid	Simple + VP layouts
Hash	Hybrid layout with hash values for nodes and a separate table for hash-to-node mappings

stored. The output of this phase is a key-value pair based on the computation that is performed. (ii) An intermediate **Sort** phase in which the output of the Map phase is sorted based on keys. (iii) A **Reduce** phase in which a reducer aggregates the values for a common key and then further processes them before producing the desired result.

2.5 Overview of HBase

Apache HBase[5] is an open source column-oriented store that is modeled after Google's BigTable [9]. HBase uses the HDFS as the underlying storage mechanism for tables that are created in HBase. Hadoop and HBase allow the use of HBase tables as sources and sinks for MapReduce jobs thus providing great flexibility in these jobs. Additional advantages of HBase are query predicate push down and optimizations for real time queries.

As HBase uses the HDFS architecture as the underlying storage mechanism, it inherits Hadoop's master/slave configuration. The HBase architecture consists of a single HBase master node that coordinates the distribution of data among the HBase region servers (usually one per slave). A HBase table is composed of column families and rows; these may be physically separated over the region servers. A row in a HBase table is defined by a row key. Rows in a HBase table are sorted lexicographically based on row keys. A column family contains a collection of columns and each column in a column family shares a common prefix. For example, the columns person:name and person:age are both columns of the column family person. The colon (:) character delimits a column family from a column. A column family is defined at the time of schema creation

[5]http://hbase.apache.org/

whereas columns can be declared on the fly as needed. A cell in a HBase table can then be identified as a three tuple: {row, column, version}. The version dimension of a tuple allows a user to store multiple versions of a column value for the same row value. Finally, HBase requires the periodic use of a "commit" operation to reflect any changes (*viz.* inserts, deletes or updates) to an HBase table.

2.6 Overview of Storm

Storm[6] is an open source distributed realtime computation system. Storm is similar to Hadoop in the sense that it provides users with a general framework for performing computations in realtime, much like Hadoop provides users with a general framework for performing batch processing operations. Storm provides the following key properties: (i) Support for a broad range of use cases such as stream processing and continuous computation. (ii) Storm is scalable and has the ability to process massive numbers of messages per second. (iii) Storm guarantees that every message will be processed and thereby ensures that there is no data loss. (iv) Storm clusters are easily manageable and extremely robust. (v) Storm ensures fault-tolerance by automatically reassigning tasks that fail during execution. (vi) Storm's components are programming language agnostic and therefore Storm can be utilized by nearly anyone.

A Storm cluster is made up of two kinds of nodes: the **master** node and the **worker** nodes. The master node runs a daemon called **"Nimbus"** that is responsible for distributing code around the cluster, assigning tasks to machines and monitoring for failures. Every worker node runs a daemon called **"Supervisor"** that listens for work assigned to it by Nimbus and starts/stops worker processes to accomplish this work. The coordination between Nimbus and Supervisors is done through a Zookeeper cluster.

Storm uses the concept of a **"topology"** to perform realtime computation. A Storm topology is analogous to a MapReduce job, however, a key difference is that a MapReduce job eventually finishes while a topology runs forever or until it is killed. A topology is a directed graph of spouts and bolts that are connected together with stream groupings. Note that the **stream** is the core abstraction in Storm and represents an unbounded sequence of tuples that is created and processed in a parallel fashion using a distributed cluster. A **spout** acts as a source of streams for a topology and usually reads tuples from an external source and emits them into the topology. A **bolt** is used to perform the desired processing within a topology such as initiating connections with a database, performing operations such as filtering, aggregations and joins. A **stream grouping** defines how a stream that is being input to a particular bolt is partitioned between that bolt's tasks. Finally, Storm guarantees that every spout tuple will be fully processed by the topology and also provides several configurations for customizing the behavior of the nimbus, supervisors and running topologies.

[6] https://github.com/nathanmarz/storm

Chapter 3

StormRider Architecture

In this chapter we begin by presenting an overview of the architecture employed by **StormRider**. This is followed by a detailed description of the various operations currently supported by StormRider for interacting with social networks.

3.1 Architectural Overview of StormRider

StormRider offers a simple abstraction for the different ways in which a user can interact with a social network, as its central internal interface (**Model-SN Interface**). This abstraction can then be used to store, query and analyze different social networks. The main contribution of StormRider is a rich API for manipulating social networks. In particular, the API provides various tools for interacting with networks, *e.g.*, storage and retrieval of social networks using various Semantic Web representations, basic social network analysis in the form of centrality estimation, *etc.* The key architectural goals of StormRider are:

- The ability for an application programmer to be able to store and analyze networks of their choice by implementing simple interfaces.
- Allow easy access to a few fundamental social network analysis metrics such as centrality computation.
- Open up the Semantic Web space for storage and retrieval of social networks, thereby allowing reasoning algorithms to be applied to them so that they may reveal richer patterns that were previously undetectable.

Figure 3.1 presents an architectural overview of StormRider. User applications (the **Application** block) can interact with an abstract social network Model (the **Model-SN** block) which translates high-level user-defined operations (*viz.* store, query and analyze) on a social network into low-level operations on the underlying representations used for that network by StormRider. The low-level operations are implemented in Storm (**Add-Topology**, **Query-Topology** and **Analyze-Topology** blocks) and are designed as such to support

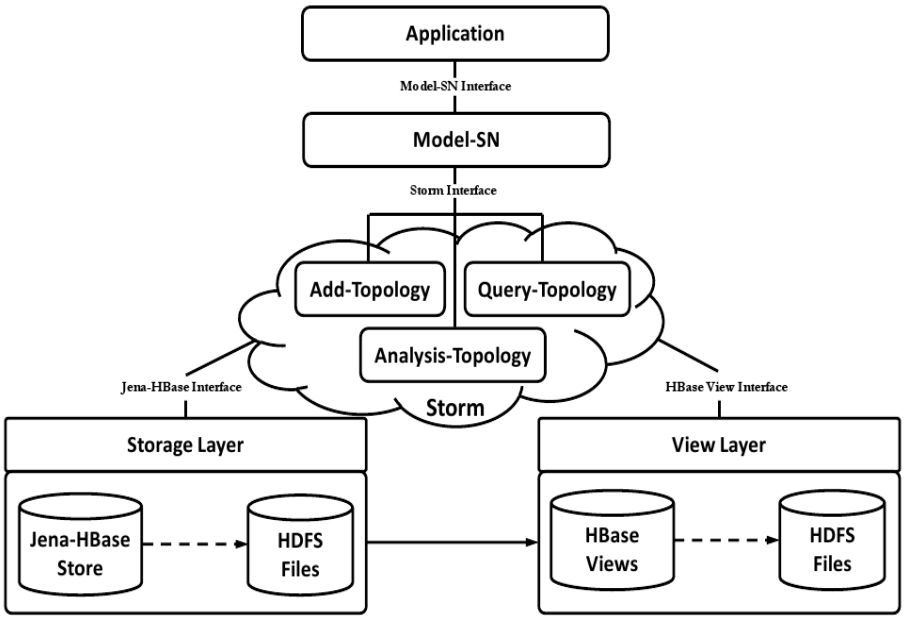

Figure 3.1: StormRider - Architectural Overview

evolving social networks. However, applications can also use these same topologies to work with single snapshots of a network. The Storm framework then internally interfaces with the **Storage Layer** (Jena-HBase), through the **Jena-HBase Interface**, and the **View Layer** (HBase tables used as materialized views), through the **HBase View Interface**, to execute topologies on underlying networks stored in them. We next present a detailed description of the Storage, View and Topology layers.

3.2 Storage Layer

This layer, comprising of Jena-HBase, is used to store networks in a RDF representation in a cloud-based framework. Since Jena-HBase simply provides a HBase-backed storage subsystem for the Jena framework, it supports all the functionalities provided by Jena for building and querying RDF graphs. The storage of networks in RDF when combined with ontologies defined in other languages (*viz.* RDFS and OWL) opens up the possibility of using reasoning algorithms on the networks to infer previously hidden information from them. The use of a cloud-based framework provides advantages such as scalability, reliability, fault-tolerance, *etc.* Recall that we provided a brief summary of the Jena-HBase framework in Section 2.3, for additional details an interested reader is referred to our detailed technical report [7].

3.3 View Layer

This layer comprises of HBase tables that store metadata about nodes in a network. The metadata is mainly used to facilitate a speed-up in performance during the analysis of a network. In StormRider, we currently focus on centrality estimation, particularly degree, closeness and betweenness, as a part of analyzing a social network. Since both, closeness and betweenness centrality are path-based metrics, we make use of a variant of the *landmark-based* technique [10, 11] for fast shortest path distance estimation as a part of the analysis topology implemented in Storm. We present below a brief overview of the *landmark-based* technique and our adaptation of it, followed by a detailed description of the HBase views.

The *landmark-based* technique for shortest path estimation involves selecting a set of landmark nodes and computing the shortest distance from all other nodes to these nodes offline [10]. The idea is to represent each non-landmark node as a vector of shortest path distances to the set of landmarks. Then at runtime, when the shortest distance between a pair of nodes is needed, it is estimated by using these precomputed vectors. Additionally, in reference [10], the authors prove that the landmark selection problem is NP-Hard and they subsequently present different landmark-selection strategies such as Random, Degree and Centrality, along with constrained and partitioned variants of them.

In StormRider, we have made the following change to the idea of maintaining a vector of shortest path distances to landmark nodes: For every non-landmark node, we simply store its nearest landmark along with the shortest distance to that landmark rather than a vector of distances to all landmark nodes. This simplification allows us to better manage shortest paths when a network is evolving over time, while maintaining relatively accurate results for shortest path estimation between any two nodes. Additionally, we also use the Degree strategy as our landmark-selection strategy since it strikes the right balance between approximation quality and computational efficiency.

Figures 3.2 and 3.3 below present the Node-centric and Landmark-centric views that are constructed for each network link that a user wants to analyze, where the corresponding network containing that link is stored in Jena-HBase. Also, note that in the figures below the values in the first column (NodeId and LandmarkId&NodeId) are used as row keys in the actual implementation of the views as HBase tables.

Figure 3.2 presents an example of a Node-centric view. The idea of this view is to present metadata for every node of the network as a distinct tuple in the underlying HBase table. Each node in the network is represented by assigning its unique identifier to a distinct row key (*viz.* NodeId) in the table. This metadata includes the following:

- Adjacency List information: This column family stores the adjacency list of every node in the network.
- Metric information: This column family stores information about the various metrics (*viz.* degree, closeness and betweenness centrality) currently

Node Id	Adj-List	Metric			Landmarks		
		DegC	CloseC	BetC	Is-Landmark	Dist-To-Closest-Landmark	Closest-Landmark
sub1	sub2 sub3	0.75	0.5	0.5	Y	0	sub1
sub10	sub20 sub30	0.1	0.03	0.02	N	1	sub1

Figure 3.2: HBase View: Node-centric information

supported by StormRider for every node in the network. Each metric (*viz.* DegC, CloseC and BetC) is represented as a separate column under the "metric" column family and stores the current value of that metric for a given node.

- Landmark information: This column family stores landmark-related information for a given node. This includes information as to whether a given node is a landmark node (Is-Landmark column), the distance from a given node to its nearest landmark node (Dist-To-Closest-Landmark column) and a given node's closest landmark node (Closest-Landmark column).

LandmarkId&NodeId	Nodes		
	Distance	Num-Of-Paths	Paths
sub1&sub10	2	1	sub1-sub2-sub10
sub1&sub50	2	2	sub1-sub5-sub50 sub1-sub25-sub50

Figure 3.3: HBase View: Landmark-centric information

Figure 3.3 presents an example of a Landmark-centric view. The idea of this view is to present information that is specific to each of the nodes that have been currently designated as landmark nodes. Note that the row key in this view is to be interpreted as follows: The first part of the key (*viz.* LandmarkId) denotes a designated landmark node, while the second part of the key (*viz.* NodeId) denotes a non-landmark node that is connected to the given landmark

node through some path. Additionally, every tuple of this view represents path-related information on how a non-landmark node (represented by the NodeId part) is connected to a landmark node (represented by the LandmarkId part). This path-related information includes the following:

- Node-specific information: This column family stores path-related information for paths from a given non-landmark node to a designated landmark node. This includes information about the number of hops between a given non-landmark node and a given landmark node (Distance column), the number of distinct paths between the two nodes (Num-Of-Paths column) and the actual paths, enumerated as a sequence of nodes, between them (Paths column).

3.4 Topology Layer

As stated earlier, the Topology layer comprises of various topologies through which a user interacts with an evolving network. Note that a user needs to implement application-specific topologies based on their requirements for interacting with networks. Then, StormRider's API executes these topologies on behalf of users. Thus, StormRider can execute topologies that interact with any network. However, to demonstrate the efficacy of StormRider, we have used the Twitter network as the basis for the construction of sample Add-, Query- and Analyze-Topologies, each of which is described next.

3.4.1 Add-Topology for Twitter Dataset

An Add-Topology allows a user to store a network of their choice in the underlying Jena-HBase framework. Since StormRider only provides a sample Add-Topology for the Twitter network, application programmers need to define their own Add-Topologies for storing social networks of their choice. StormRider's API then executes these topologies on behalf of the users.

Figure 3.4 shows the design of a sample Add-Topology that adds Twitter data to the StormRider framework. The general idea is to have a spout that reads data from the Twitter API and then uses it for multiple purposes. The data is converted into an RDF representation and stored in Jena-HBase (denoted by the **Storage** block). At the same time, the data is also used to update the views (denoted by the **Node-centric View** and **Landmark-Information** blocks) maintained in the view layer. Note that in this work we only analyze the "friendship" link between Twitter users. Therefore, both views are created/maintained only for this link. We have divided the discussion about the Add-Topology into two parts, namely the spout that reads Twitter data and the various bolts that update the Node-centric and Landmark-centric views. Additionally, to have a clear understanding of how our system works, we have presented the discussions on the spout as well as the bolts as a series of algorithms, followed by detailed explanations. Before we begin this discussion, we give some notes that apply to all algorithms presented throughout this chapter.

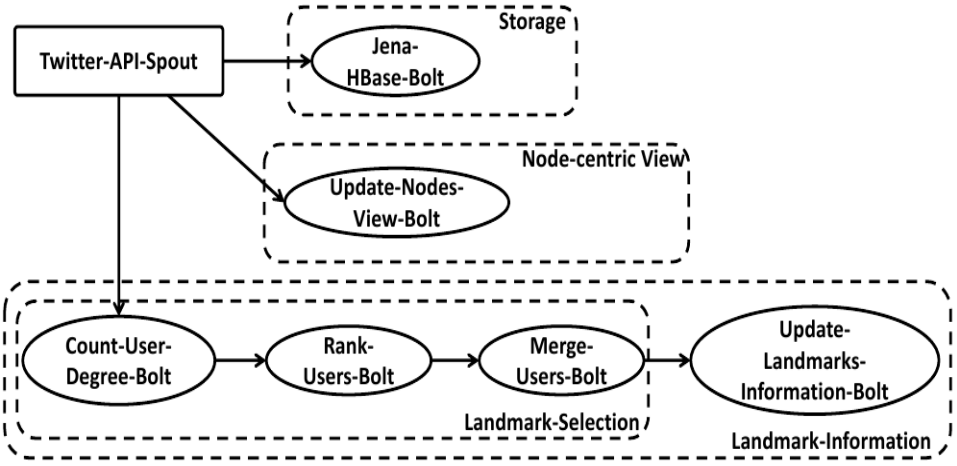

Figure 3.4: A sample Add-Topology for the Twitter network

Algorithmic Notes: All the pseudocode given in the algorithms below is executed as a particular method, *viz.*, `nextTuple()` in a spout or `execute(Tuple input)` in a bolt. Additionally, for algorithms that describe bolts, any input parameter other than a *tuple*, is always constructed/initialized by the `prepare()` method of a bolt. Therefore, these additional parameters are simply shown as inputs to the algorithms for simplicity and convenience.

Add-Topology - Spout

As stated earlier, a Storm spout, **Twitter-API-Spout**, is used to read data from the Twitter API, which is then injected into various bolts that update the storage and view layers. We begin by presenting a description of the Twitter-API-Spout followed by explanations for the different bolts that update the storage and view layers.

Algorithm 1 presents a set of steps used by the Twitter-API-Spout to read data from Twitter and then inject it into the various bolts used to update the storage and view layers in StormRider. The algorithm iterates in an infinite loop that keeps adding/updating user information in the system (lines 1-13). In this loop, the algorithm first selects a random user from the Twitter network (line 2). Algorithm 1 then retrieves information specific to the selected user such as name, friends, *etc.* (line 3), and then emits a tuple for each piece of information to the **Jena-HBase-Bolt** (lines 4-6). After this process is complete, the algorithm emits a separate tuple containing only the current user's identifier to the **Update-Nodes-View-Bolt** (line 7). Finally, for every user stored in the system, the algorithm repeatedly emits a tuple containing that user's identifier to the **Count-User-Degree-Bolt** (lines 8-11). The tuple is emitted once for every "friend" of the current user. To perform this task in reality, the algorithm

Algorithm 1 TWITTER-API-SPOUT()

Input: Q, M

1: **while** *true* **do**
2: Pick a random Twitter user
3: Get information specific to this user such as name, location, tweets, friends, followers *etc.* using the Twitter API
4: **for** each piece of information **do**
5: Emit a tuple containing that information, in the form (user-identifier, property, value), to the Jena-HBase-Bolt
6: **end for**
7: Emit a tuple with the user-identifier to the Update-Nodes-View-Bolt
8: $usersAndNeighbors \leftarrow execute(Q, M)$
9: **for** $userAndNeighbor \in usersAndNeighbors$ **do**
10: Emit a tuple containing the user-identifier in $userAndNeighbor$
11: **end for**
12: **end while**

executes the following SPARQL query Q over the underlying Jena-HBase model M,

```
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x ?y
WHERE { ?x twitter:Has_Friend ?y }
```

and emits a new tuple for each unique binding of values to variables `?x` and `?y`.

Add-Topology - Bolts

We now present a detailed discussion of the various bolts implemented as a part of the Add-Topology that are used to update the storage and view layers.

The Storage layer Bolt: This layer consists of a single bolt, **Jena-HBase-Bolt**, that simply takes tuples containing user-specific information that are emitted from the Twitter-API-Spout, converts them into a RDF representation (*viz.* triples), and then stores these triples in an underlying Jena-HBase model.

Algorithm 2 JENA-HBASE-BOLT()

Input: *tuple*, M

1: $subject \leftarrow tuple.getString(0)$
2: $predicate \leftarrow tuple.getString(1)$
3: $object \leftarrow tuple.getString(2)$
4: $M.add(triple(subject, predicate, object))$

Algorithm 2 presents a set of steps used by the Jena-HBase-Bolt to add Twitter data to an underlying Jena-HBase model. For every tuple received by

Jena-HBase-Bolt, the algorithm extracts a string representation of the three constituent parts of this tuple (lines 1-3). The three components correspond to the *subject*, *predicate* and *object* respectively of the triple that can be generated for the given input tuple. The algorithm first constructs this corresponding triple using these components, and then adds the triple to the underlying Jena-HBase model M (line 4).

The View layer Bolts: This layer consists of multiple bolts that are used to update the Node-centric and the Landmark-centric view with metadata pertaining to the nodes of the Twitter network. The **Update-Nodes-View-Bolt** takes the identifier of the user currently selected by the Twitter-API-Spout as an input, and updates the Node-centric View with metadata related to this user. Note that this bolt only updates information for non-landmark nodes, while the **Landmark-Information** block updates information associated with landmarks. The sequence of steps used by this bolt to perform this task is presented below in Algorithm 3.

Algorithm 3 UPDATE-NODES-VIEW-BOLT()

Input: *tuple*, Q, M, HBase-Nodes-View

1: $node \leftarrow tuple.getString(0)$
2: $isLandmarkNode \leftarrow lookup(node,$ HBase-Nodes-View$)$
3: **if** $isLandmarkNode == false$ **then**
4: $adjList \leftarrow execute(Q, M)$
5: HBase-Nodes-View.$add(node,$ Adj-List$, adjList)$
6: closest-landmark $\leftarrow BFS(node)$
7: HBase-Nodes-View.$add(node,$ Is-Landmark, N$)$
8: HBase-Nodes-View.$add(node,$ Dist-To-Closest-Landmark, closest-landmark.$distance$$)$
9: HBase-Nodes-View.$add(node,$ Closest-Landmark, closest-landmark.$node$$)$
10: **end if**

The algorithm begins by extracting a string representation of the node currently under consideration by the Twitter-API-Spout into the variable *node* (line 1). The algorithm next looks-up the Node-centric view and determines if the current *node* is a landmark node (line 2). If it is not, then the algorithm updates the Node-centric view with metadata related with the current non-landmark node. The algorithm first queries model M using the following query Q,

```
PREFIX ex: <http://www.example.org/twitter#>
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?y
WHERE { ex:node twitter:Has_Friend ?y }
```

that retrieves all other Twitter users connected to the current user, `<ex:node>`, by a "friendship" link (line 4). This list of users, denoted by the variable $adjList$, is then added to the "Adj-List" column family in the Node-centric view for the current user (line 5). The algorithm then determines the closest landmark by

running the breadth-first-search (BFS) algorithm from the current *node* (line 6). The result of the BFS algorithm is stored in a data structure, "closest-landmark", by Algorithm 3. This data structure stores both, the distance to the closest landmark (in closest-landmark.*distance*) as well as the user-identifier of the closest landmark node (in closest-landmark.*node*). Next, the algorithm adds/updates the Node-centric view with this new information (lines 7-9). In particular, the algorithm adds/updates the following column families in the Node-centric view: (i) "Is-Landmark" is updated with the value "N" since the current node is not a landmark node. (ii) "Dist-To-Closest-Landmark" is updated with the value in closest-landmark.*distance*. (iii) "Closest-Landmark" is updated with the user-identifier in closest-landmark.*node*.

The **Landmark-Information** block in Figure 3.4 updates information associated with landmarks. This block consists of multiple components, each of which is described below in greater detail.

The **Landmark-Selection** block is used to select the top-k nodes from amongst all the nodes present in the system. The top-k nodes are selected using the Degree strategy given in [10] and are stored as a list of landmark nodes. Note that multiple executions of the Landmark-Selection block could lead to changes to this list. The first bolt in this block, **Count-User-Degree-Bolt**, takes as input a tuple consisting of a user-identifier and uses this information to compute a count of all friends (*i.e.* degree for the "friendship" link) for a given user. Algorithm 4 presents the sequence of steps required to perform this operation. This bolt makes use of a field grouping on the user-identifier field to ensure that tuples for a given user, emitted by the Twitter-API-Spout, always go to the same bolt task.

Algorithm 4 COUNT-USER-DEGREE-BOLT()

Input: $tuple$, $hashMap < userId, Long >$

1: $node \leftarrow tuple.getString(0)$
2: $deg \leftarrow hashMap.get(node)$
3: **if** $deg == null$ **then**
4: $\quad hashMap.put(node, 1)$
5: **else**
6: $\quad hashMap.put(node, deg + 1)$
7: **end if**
8: $opTuple.add(node, hashMap.get(node))$
9: Emit $opTuple$

Algorithm 4 begins by extracting a string representation of the user-identifier from the incoming tuple (line 1). The algorithm then looks-up this identifier in the hash-map, and tries to retrieve the current user's degree (line 2). If the degree is null, the user does not exist in the hash-map and is therefore added into it with a degree of 1 (line 4). On the other hand, if the user does exist in the hash-map, that user's degree is simply incremented by 1 (line 6). Finally, a tuple consisting of the current node and it's updated degree is then forwarded

to the Rank-Users-Bolt (lines 8-9).

The remaining bolts in the Landmark-Selection block are part of a design pattern specified in Storm that allows us to select the top-k landmarks based on the degrees of all nodes in the network. The broad idea is to divide the task of selecting the top-k landmarks between two bolts as follows: (i) The first bolt finds the top-k nodes within separate partitions of the stream. (ii) The second bolt then merges the different top-k lists into a global list of top-k landmark nodes. This approach ensures a scalable solution even in the presence of very large streams.

The **Rank-Users-Bolt** is used to find the top-k landmark nodes within separate partitions of the stream. The sequence of steps used to perform this task is outlined below in Algorithm 5. This bolt also makes use of a field grouping on the user-identifier of a node so that tuples for the same node are always sent to the same bolt task.

Algorithm 5 RANK-USERS-BOLT()

Input: *tuple*, k, *list*

1: $node \leftarrow tuple.getString(0)$
2: $nodePos \leftarrow list.find(node)$
3: **if** $nodePos == null$ **then**
4: $\quad list.add(tuple)$
5: **else**
6: $\quad list.add(nodePos, tuple)$
7: **end if**
8: Sort *list* in decreasing order based on the degrees of nodes
9: **if** $list.size > k$ **then**
10: $\quad$ Remove *tuple* at position k
11: **end if**
12: $opTuple.add(list)$
13: Emit *opTuple*

Algorithm 5 receives as input a tuple consisting of a user-identifier for a node and the node's current count of "friends". Additionally, the algorithm also receives as input, the value k which denotes an upper bound for the number of users to be used as landmark nodes, and a *list* that holds the top-k landmark nodes. Algorithm 5 begins by extracting the user-identifier from the tuple (line 1). Next, the algorithm tries to find the position, *nodePos*, of the node in *list* (line 2). If the node is not part of *list*, the tuple containing the node is added to the end of *list*, while if the node is present, the tuple is added to *list* at the position found in *nodePos* (lines 3-7). The algorithm then sorts the tuples in *list* in decreasing order based on the degrees (*viz.* number of friends) associated with nodes (line 8). If the size of *list* exceeds the upper bound k, the tuple at position k is removed since at any given time the maximum size of *list* can be $k + 1$ (lines 9-11). Finally, the algorithm constructs a new tuple consisting of the updated *list* and emits this tuple to the Merge-Users-Bolt (lines 12-13).

The **Merge-Users-Bolt** is used to merge the various lists of top-k landmark nodes that were generated by the Rank-Users-Bolt into a global list of top-k landmark nodes. The pseudocode used to perform this task is given below in Algorithm 6. This bolt makes use of a global grouping on the lists generated by the Rank-Users-Bolt to ensure that all top-k lists are sent to a single bolt task.

Algorithm 6 MERGE-USERS-BOLT()

Input: *tuple*, k, *list*
Output: *opTuple*

```
1:  nodeList ← tuple.getString(0)
2:  for l ∈ nodeList do
3:      nodePos ← list.find(l)
4:      if nodePos == null then
5:          list.add(l)
6:      else
7:          list.add(nodePos, l)
8:      end if
9:      Sort list in decreasing order based on the degrees of nodes
10:     if list.size > k then
11:         Remove elements in list for k ≤ i < list.size()
12:     end if
13: end for
14: opTuple.add(list)
15: Emit opTuple
```

Algorithm 6 receives as input a tuple consisting of some top-k list generated by one of the tasks of Rank-Users-Bolt. Additionally, the algorithm also receives as input, the value k which denotes an upper bound for the number of users to be used in the global list of landmark nodes, and a *list* that holds the global top-k landmark nodes. Algorithm 6 begins by extracting the list, *nodeList*, from the tuple (line 1). Then, for every node l in *nodeList*, the algorithm repeats a sequence of steps (lines 3-12), similar to the steps used in Algorithm 5, to update *list*, the global list of top-k landmark nodes. The only difference between these sequences of steps is that in this case, the size of *list* could be greater than $k+1$, therefore, the algorithm now removes all nodes beginning with position k all the way through to position $list.size() - 1$ (line 11). Finally, the algorithm constructs a new tuple consisting of the *list* of global top-k landmark nodes and emits this tuple to the Update-Landmarks-View-Bolt (lines 14-15).

The **Update-Landmarks-Information-Bolt** is used to update the Landmark-centric view with metadata regarding shortest path information on how non-landmark nodes are connected with landmark nodes. Additionally, this bolt also updates the Node-centric view with metadata pertaining to each landmark in the top-k list. Algorithm 7 presents a list of successive steps that can be used to perform these tasks.

Algorithm 7 UPDATE-LANDMARKS-INFORMATION-BOLT()

Input: *tuple*, Q, M, *nonLandmarksList*, HBase-Nodes-View, HBase-Landmarks-View

```
1: list ← tuple.get(0)
2: for landmark ∈ list do
3:    Clear rows in HBase-Landmarks-View that contain landmark
4:    landmarkInfo ← SSSP(landmark)
5:    for node ∈ nonLandmarksList do
6:       nodeInfo ← landmarkInfo.find(node)
7:       HBase-Landmarks-View.add(landmark&node, Distance, nodeInfo.distance)
8:       HBase-Landmarks-View.add(landmark&node, Num-Of-Paths, nodeInfo.numOfPaths)
9:       for i = 1 to numOfPaths do
10:         HBase-Landmarks-View.add(landmark&node, Paths, nodeInfo.getPath(i))
11:      end for
12:   end for
13:   adjList ← execute(Q, M)
14:   HBase-Nodes-View.add(node, Adj-List, adjList)
15:   HBase-Nodes-View.add(landmark, Is-Landmark, Y)
16:   HBase-Nodes-View.add(landmark, Dist-To-Closest-Landmark, 0)
17:   HBase-Nodes-View.add(landmark, Closest-Landmark, landmark)
18: end for
```

Algorithm 7 receives as input a tuple containing a list of the global top-k landmark nodes. The algorithm also receives as input the query Q that was used earlier to retrieve the adjacency list of a given node. Further, Algorithm 7 receives a Jena-HBase model M that stores data for the network under consideration. Finally, the algorithm also receives the list of non-landmark nodes, namely *nonLandmarksList*, as well as the Node-centric and Landmark-centric views, namely HBase-Nodes-View and HBase-Landmarks-View as inputs. Note that *nonLandmarksList* can be generated by querying the Node-centric view for all nodes that contain a value "N" for the "Is-Landmark" column family. The algorithm begins by extracting the global list of top-k landmarks from the input tuple (line 1). For every landmark node in this list, *landmark*, the algorithm first clears the rows in HBase-Landmarks-View that contain *landmark* (line 3). Then, the algorithm executes the single-source-shortest-path (SSSP) algorithm for *landmark* (line 4). This algorithm computes the shortest paths from *landmark* to all other nodes in the network and stores the results in a data structure, *landmarkInfo*. The algorithm then iterates over *nonLandmarksList* and in each iteration, the algorithm first finds path-related information for the current node (*node*) which is then stored in a node-specific data structure, *nodeInfo* (line 6). Algorithm 7 then updates HBase-Landmarks-View with path-related information for *node* (lines 7-11). This information includes the following: (i) The distance between *landmark* and *node* is stored in the column family "Distance". (ii) The number of paths between *landmark* and

node are stored in the column family "Num-Of-Paths". (iii) The actual paths between *landmark* and *node* are stored as sequences of nodes in the column family "Paths". Finally, the algorithm adds/updates the Node-centric view with information pertaining to *landmark* (lines 13-17). In particular, the algorithm adds/updates the following column families in the Node-centric view: (i) "Adj-List" is updated with the adjacency list of *landmark*, which is obtained by executing query Q on M. (ii) "Is-Landmark" is updated with the value "Y" since the current node is a landmark node. (iii) "Dist-To-Closest-Landmark" is updated with the value "0". (iv) "Closest-Landmark" is updated with the value *landmark*, since the node under consideration is itself a landmark node.

As stated earlier, we have only presented a sample Add-Topology for the Twitter dataset. Application programmers wishing to use StormRider need to implement their own topologies for storing networks of their choice. However, they can reuse the spouts and bolts that are implemented in StormRider within their custom Add-Topologies. We now move on to a detailed discussion of the Query-Topology that we have implemented for the Twitter dataset.

3.4.2 Query-Topology for Twitter Dataset

A Query-Topology allows users to execute queries (continuous or ad-hoc) on a network stored in the underlying storage and view layers as the data from that network is being streamed into Jena-HBase. Note that StormRider's existing Query-Topology can be used to execute SPARQL queries on any of the underlying networks. Additionally, application developers can implement their own Query-Topologies for querying networks stored in the underlying storage and view layers. In this way, StormRider allows various types of query topologies to be executed over the data in the storage and view layers. We now present a few samples of the types of topologies that can be currently handled by StormRider.

SPARQL Query-Topology

A SPARQL Query-Topology is used to execute a user-defined SPARQL query (continuous or ad-hoc) over the RDF data stored in Jena-HBase. A Query-Topology that executes a SPARQL query, called SPARQL-Query-Topology in StormRider, consists of a spout called **SPARQL-Query-Spout** that takes the input SPARQL query and executes it over the desired network stored in Jena-HBase. Then, the **HBase-Result-Table-Loader-Bolt** is used to load the results into a user-specified HBase table, after which these results can be streamed to a client. Figure 3.5 depicts the execution strategy of the SPARQL-Query-Topology.

We now present some sample SPARQL queries as well as the pseudocode used by the SPARQL-Query-Spout to execute them. These queries represent the ability of StormRider to not only support general-purpose SPARQL queries but also special-purpose use cases such as automatically querying evolving networks as well as being able to retain and access prior states of a network.

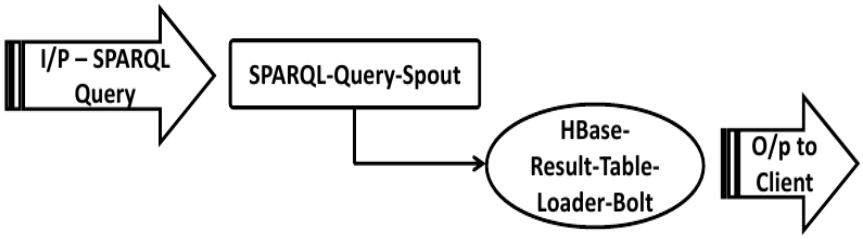

Figure 3.5: A Query-Topology used to execute SPARQL queries

Example 1: Consider the following simple SPARQL query that aims to find all Twitter users who have listed their current location as "Richardson, TX". This type of query can be classified as an ad-hoc query, since it will most likely be executed only when some user wants to find out about other Twitter users that live in Richardson, TX, which will usually be an infrequent request.

```
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x
WHERE { ?x twitter:Location "Richardson, TX" }
```

Example 2: Now consider a slightly more complex use case that aims at tracking the neighborhood of a given Twitter user. These types of requests are common in tasks such as police surveillance and targeted advertising. The idea behind this type of query is to track the evolution of a user's neighborhood over time and is therefore very significant in scope. The use of Semantic Web technologies in StormRider easily allows us to accomplish this task through the use of SPARQL path queries. This type of query can be classified as a continuous query since a user would want to track another user's neighborhood over an extended period of time. Furthermore, such a query requires tracking the various paths that emerge from the node under consideration and as such can be effectively captured by a regular-expression SPARQL query. Towards this end, we make use of the Gleen library [12] to define such regular-expression SPARQL queries. This library is closely integrated with the Jena framework and extends SPARQL with support for defining regular path queries over Jena models. Since Jena-HBase simply acts as an HBase-backed storage subsystem for the Jena framework, we can directly use Gleen to define regular path queries over models stored in Jena-HBase. The following query sample can be used to track the neighborhood of a Twitter user "John" covering his friends as well as friends-of-his-friends (2 hops). The SPARQL-Query-Topology will automatically execute this query periodically so that the neighborhood of the user "John" is tracked over an extended period of time. Note that 2 hops is simply used for illustrative purposes, and in practice, StormRider can support queries with an arbitrary number of hops.

```
PREFIX ex: <http://www.example.org/twitter#>
```

```
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x
WHERE { <ex:John> gleen:onPath("([twitter:Has_Friend]/[twitter:Has_Friend])" ?x) }
```

Example 3: We now present a use case that aims at allowing users to store and access all versions of an evolving network. This request is relevant for intelligence agencies that want to trace the evolution of a potential terrorist cell. The idea of storing and accessing prior snapshots is similar in spirit to the *ImmortalDB* project which builds transaction time database support into the SQL Server Engine [13]. The implementation of this feature in StormRider requires no changes to the actual framework since it is automatically enabled for us through the use of a RDF representation and the SPARQL query language. However, the way in which a user interacts (*viz.* stores and accesses) with a network needs to be slightly modified as given below.

Timestamping and Version management: Since StormRider uses a RDF representation to store network data, a triple t_i can be reified with a timestamp value T_i that denotes the time when the triple was constructed after the discovery of its constituent parts (*viz.* subject, predicate and object) through the Twitter API. Any subsequent changes to t_i (*e.g.* re-discovery of the triple or updates to a triple) lead to the insertion of a new triple, t_j, reified with timestamp T_j indicating the construction time of t_j. Note that, $T_j > T_i$, since t_j was discovered at a later time than t_i. In this way, StormRider stores two different versions of the triple, namely t_i and t_j, which can be subsequently accessed with the appropriate timestamp (T_i or T_j). Similarly, when a triple needs to be deleted, a special version of the triple is inserted which is reified with a timestamp that denotes when the triple was deleted. Additionally, the task of version control does not require any special-purpose functions. On the contrary, all versions of a triple are stored and maintained within the same Jena-HBase model. Consider the following example in which StormRider discovers that a user "John" has a friend "James" through the Twitter API. Note that the following prefixes are used in the subsequent RDF representation,

```
ex: <http://www.example.org/twitter#>
twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
```

This relationship is captured in the following triple,

```
ex:John twitter:Has_Friend ex:James
```

which is then stored in a Jena-HBase model. Additionally, the triple is reified with the timestamp value of when the triple was discovered. This leads to the addition of the following triples to the same model,

```
ex:John twitter:Has_Friend ex:James
ex:triple123 rdf:type rdf:Statement
ex:triple123 rdf:subject ex:John
ex:triple123 rdf:predicate twitter:Has_Friend
ex:triple123 rdf:object ex:James
ex:triple123 twitter:Timestamp "2012-01-25T21:00:00"^^xsd:dateTime
```

Suppose that at a later point in time we now discover that the same user "John" now has two friends, namely "James" (as before) and "Joe". The re-discovery of John's relationship with James and the discovery of a new relationship between John and Joe leads to the addition of the following triples to the model,

```
ex:John twitter:Has_Friend ex:James
ex:triple123 rdf:type rdf:Statement
ex:triple123 rdf:subject ex:John
ex:triple123 rdf:predicate twitter:Has_Friend
ex:triple123 rdf:object ex:James
ex:triple123 twitter:Timestamp "2012-01-25T21:00:00"^^xsd:dateTime

ex:triple124 rdf:type rdf:Statement
ex:triple124 rdf:subject ex:John
ex:triple124 rdf:predicate twitter:Has_Friend
ex:triple124 rdf:object ex:James
ex:triple124 twitter:Timestamp "2012-01-26T22:00:00"^^xsd:dateTime

ex:John twitter:Has_Friend ex:Joe
ex:triple125 rdf:type rdf:Statement
ex:triple125 rdf:subject ex:John
ex:triple125 rdf:predicate twitter:Has_Friend
ex:triple125 rdf:object ex:Joe
ex:triple125 twitter:Timestamp "2012-01-26T22:00:00"^^xsd:dateTime
```

To enable this feature of historical storage of network data in StormRider, a user simply needs to reify the triples that are being added to the underlying Jena-HBase models in their custom-built Add-Topologies.

SPARQL syntax: The SPARQL query language itself does not need to be extended in any way to provide access to historical versions of a network in StormRider. A user can query older snapshots of the network by making use of the FILTER construct of SPARQL with appropriate timestamp values. This is in sharp contrast to *ImmortalDB* in which changes were made to the data definition language to include an Immortal attribute that allows the definition of transaction-time tables. Additionally, *ImmortalDB* supported historical queries through the use of an "AS OF" clause with the "Begin Transaction" statement. For example, the following SPARQL query could be used to find users who were connected with the user "John" through the "friendship" link at any point in time prior to and including a timestamp value of `"2012-01-25T21:00:00"^^xsd:dateTime`,

```
PREFIX ex: <http://www.example.org/twitter#>
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?z
WHERE
{
     ?x twitter:Timestamp ?y .
     ?x rdf:subject <ex:John> .
     ?x rdf:predicate <twitter:Has_Friend> .
```

```
     ?x rdf:object ?z .
     FILTER( ?y <= "2012-01-25T21:00:00"^^xsd:dateTime )
}
```

Thus, to enable historical queries of network data, a user only has to define the appropriate SPARQL query that makes use of a FILTER construct along with an appropriate timestamp value that captures the time period they want to query.

Example 4: Consider a use case that allows users to continuously search and monitor publicly available information posted to online social media websites. This type of use case is relevant to intelligence agencies that wish to collect, analyze and share publicly available online social media information. Furthermore, the information obtained through support of such a use case allows agencies to enhance their situational awareness as well as to improve their decision making capabilities. Towards this end, several intelligence agencies (*viz.* FBI, IARPA, DARPA, *etc.*) have solicited proposals that seek to develop applications that monitor publicly available information accessible through online social media applications [14, 15, 16]. We can easily provide automated support for this use case in StormRider, which contains a Storm topology for executing continuous SPARQL queries. Additionally, we can implement a custom-built Add-Topology that provides support for monitoring publicly available information as triples are being added to Jena-HBaes. Furthermore, StormRider can support the following variations of the basic use case that was presented above:

4a: Instantly search and monitor specific keywords and strings posted to online social media websites. For example, the following SPARQL query can be used to continuously search for Twitter users that post tweets containing the keyword "bombing".

```
PREFIX ex: <http://www.example.org/twitter#>
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x ?y
WHERE
{
     ?x tw:Has_Tweet ?y
     ?y tw:Tweet_Text ?z
     FILTER regex(?z, "bombing", "i")
}
```

4b: Simultaneously conduct multiple word combination searches that monitor numerous keywords at the same time. For example, the following SPARQL query can be used to continuously search for Twitter users that post tweets containing the keyword "bombing" or "explosion" followed by "America".

```
PREFIX ex: <http://www.example.org/twitter#>
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x ?y
WHERE
{
```

```
     ?x tw:Has_Tweet ?y
     ?y tw:Tweet_Text ?z
     FILTER regex(?z, ".*(bombing|explosion).*america.*", "i")
}
```

4c: Conduct multiple word combination searches that monitor numerous keywords for a particular location at the same time. For example, the following SPARQL query can be used to continuously search for Twitter users that post tweets with a location "Richardson,TX" and contain the keyword "bombing" or "explosion" followed by "America".

```
PREFIX ex: <http://www.example.org/twitter#>
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x ?y
WHERE
{
     ?x tw:Has_Tweet ?y
     ?y tw:Tweet_Text ?z
     ?y tw:Place_Full_Name "Richardson,TX"^^xsd:string
     FILTER regex(?z, ".*(bombing|explosion).*america.*", "i")
}
```

We now present a set of algorithms that can be used to execute any SPARQL query (continuous or ad-hoc) over any of the underlying networks stored in Jena-HBase. Algorithm 8 presents the pseudocode that is implemented as a part of SPARQL-Query-Spout given in Figure 3.5. Recall that the SPARQL-Query-Spout is used to execute a SPARQL query over a network stored in Jena-HBase. On the other hand, Algorithm 9 describes the pseudocode used as a part of HBase-Result-Table-Loader-Bolt, which loads the results of a SPARQL query, received from SPARQL-Query-Spout, into a user-specified HBase table, after which these results can be streamed to a client.

Algorithm 8 takes as input, a SPARQL query Q, a Jena-HBase model M, an *interval* (in seconds) that specifies the fixed time period after which Q needs to be executed, and the maximum number of times, $maxReports$, the user wants to execute query Q. The algorithm begins by initializing the variable $numOfReports$ to 0 (line 1) and the variable $startTime$ to the current time (line 2). The variable $numOfReports$ denotes the number of times the algorithm has executed Q and forwarded result R to HBase-Result-Table-Loader-Bolt while the variable $startTime$ denotes the time at which the algorithm began its execution. Algorithm 8 then repeats the while-do loop (lines 4-14) for a $maxReports$ number of times. Within this loop, the algorithm first updates variable $currTime$ with the current time (line 5). The algorithm then checks if an *interval* amount of time has passed since the last execution of query Q (line 6). If an *interval* amount of time has indeed passed, the algorithm re-executes query Q (line 7) and forwards the corresponding result R to HBase-Result-Table-Loader-Bolt one row, r, at a time (lines 8-10). Additionally, the algorithm also increments $numOfReports$ (line 11) and updates $startTime$ with $currTime$

Algorithm 8 SPARQL-Query-Spout()

Input: Q, M, $interval$, $maxReports$

```
1: numOfReports ← 0
2: startTime ← getCurrentTime()
3: while true do
4:    while numOfReports < maxReports do
5:       currTime ← getCurrentTime()
6:       if (currTime − startTime) ≥ interval then
7:          R ← execute(Q, M)
8:          for each result r ∈ R do
9:             Emit a tuple containing r to HBase-Result-Table-Loader-Bolt
10:         end for
11:         numOfReports ← numOfReports + 1
12:         startTime ← currTime
13:      end if
14:   end while
15: end while
```

(line 12). Finally, when the while-do loop has executed a $maxReports$ number of times, the algorithm simply loops until the Query-Topology is manually terminated.

Algorithm 9 HBase-Result-Table-Loader-Bolt()

Input: $tuple$, HBase-Result-Table

```
1: resultRow = ""
2: numOfVars ← tuple.size()
3: for i ← 0 to numOfVars − 1 do
4:    colVal ← tuple.getString(i)
5:    resultRow ← resultRow + colVal+ " "
6: end for
7: HBase-Result-Table.add(resultRow, null, null)
```

Algorithm 9 receives as its input a tuple containing a single row of the result R that was computed by Algorithm 8. Additionally, the algorithm also receives the user-specified table, HBase-Result-Table, which will store R. The algorithm begins by initializing $resultRow$ to the empty string (line 1). This variable will hold the result for row r of R generated by SPARQL-Query-Spout. Algorithm 9 then computes the number of variables, $numOfVars$, in query Q, since the input tuple will contain $numOfVars$ values in every result row r (line 2). The algorithm then iterates $numOfVars$ times to construct $resultRow$ using the values obtained from the input tuple (lines 3-6). Finally, the algorithm adds $resultRow$ as a new row in HBase-Result-Table (line 7). Then, a client application can access result R by iterating over the rows of HBase-Result-Table.

Custom-built Query-Topology

As we stated earlier, StormRider's architecture allows users to define their own custom-built Query-Topologies for query tasks specific to their own networks. In this subsection, we present an example of one such custom-built Query-Topology that we have constructed for the Twitter network. Figure 3.6 presents an overview of this topology whose goal is to periodically output the top-k users whose tweets are most often retweeted. Note that the design of this topology is similar to the one that computes the top-k landmark nodes from amongst all nodes which we presented earlier. Therefore, in this subsection we simply give a description of the topology while omitting the algorithms which are quite similar to the ones we presented earlier. We now present a detailed explanation for each of the components of this topology below.

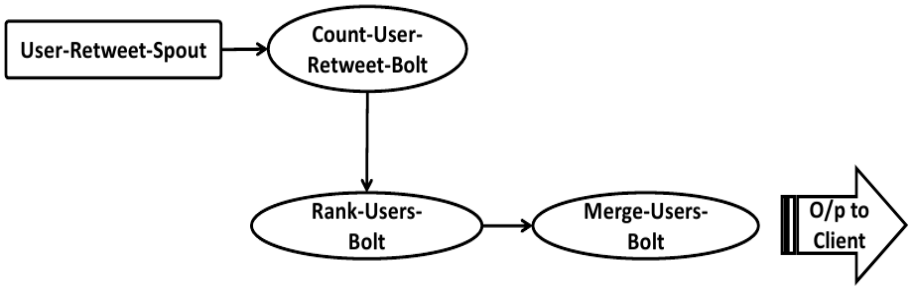

Figure 3.6: A sample user-defined Query-Topology

Figure 3.6 presents a sample topology that can be used to execute the task that we outlined above. As can be seen from the figure, the topology consists of multiple components. A Storm spout, **User-Retweet-Spout**, periodically generates tuples consisting of a Twitter user-identifier and the number of retweets for every tweet that has been posted by this user. To perform this task, the spout uses the following SPARQL query:

```
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x ?z
WHERE { ?x twitter:Has_Tweet ?y .
        ?y twitter:Tweet_Retweet_Count ?z . }
```

Every unique binding of values to variables `?x` and `?z` is used to produce a Storm tuple which is then forwarded to the Count-User-Retweet-Bolt. The **Count-User-Retweet-Bolt** simply computes a count of all retweets for a particular user-identifier that was output by the User-Retweet-Spout. Additionally, this bolt uses a field grouping on the user-identifier field to ensure that tuples with the same user-identifier always go to the same task. Finally, this bolt emits the user-identifier and the rolling count of retweets for that user as its output to the Rank-Users-Bolt. The **Rank-Users-Bolt** performs multiple parallel computations to find many top-k users across the various partitions of the stream.

This bolt also uses the same user-identifier based field grouping used earlier to ensure that tuples for the same user-identifier are sent to the same task. The different top-k lists that are generated over the entire cluster are then merged into a global top-k list using the **Merge-Users-Bolt**. This bolt makes use of a global grouping strategy to ensure that all top-k lists are sent to a single task that can then correctly merge these lists into the final top-k users list. The final top-k list is then forwarded to the client application. The rationale behind this approach is to find top-k users within separate partitions of the stream, followed by a global merging of these individual top-k's to obtain the global top-k users across the entire stream. This two-phase process ensures a scalable solution even in the presence of very large streams.

3.4.3 Analyze-Topology for Twitter Dataset

An Analyze-Topology allows users to analyze the networks that they have stored in Jena-HBase. In particular, StormRider provides a sample Analyze-Topology that performs centrality estimation on evolving social networks. Application developers can further implement their own Analyze-Topologies to analyze networks stored in the underlying storage layer. They can also make use of the metadata stored in the view layer to speed-up their analyses if they have used our custom Storm bolts to create and maintain application-specific views as a part of their Add-Topologies.

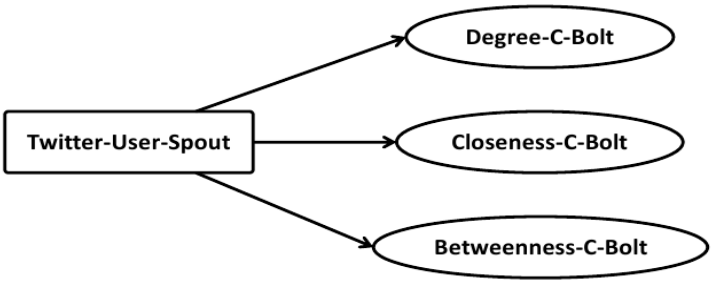

Figure 3.7: An Analyze-Topology used to compute centrality metrics on the Twitter network

Figure 3.7 shows the design of a sample Analyze-Topology that is used to compute the centralities of nodes from the Twitter network that are stored in Jena-HBase. The basic idea is to use a spout that periodically generates a list of all nodes currently stored in the system, and then emits each of these nodes as tuples to different bolts that then perform centrality estimation (*viz.* degree, closeness and betweenness) for them. We have again divided the discussion of this topology into its constituent parts, namely a spout that generates tuples containing user-identifier's of nodes, and bolts that perform centrality estimation. Additionally, we present algorithms for each of these components for a

better understanding.

Analyze-Topology - Spout

A Storm spout, **Twitter-User-Spout**, is used to periodically generate a list of all the Twitter users currently stored in the system. Then, the spout constructs a tuple for each of the users in the list which is then forwarded to the different bolts in the topology. We now present an algorithm that implements the task that we previously outlined.

Algorithm 10 TWITTER-USER-SPOUT()

Input: Q, M, *interval*

1: $startTime \leftarrow getCurrentTime()$
2: **while** *true* **do**
3: $currTime \leftarrow getCurrentTime()$
4: **if** $(currTime - startTime) \geq interval$ **then**
5: $R \leftarrow execute(Q, M)$
6: **for** $r \in R$ **do**
7: Emit a tuple containing the user-identifier in r to all bolts
8: **end for**
9: **end if**
10: **end while**

Algorithm 10 takes as input, a SPARQL query Q, a Jena-HBase model M and an *interval* (in seconds) that specifies the fixed time period after which Q needs to be executed in order to generate the list of Twitter users currently in the system. The algorithm begins by initializing the variable $startTime$ to the current time (line 1). This variable denotes the time at which the algorithm began its execution. Algorithm 10 then repeats the while-do loop (lines 3-10) forever. Within this loop, the algorithm first updates the variable $currTime$ with the current time (line 3). The algorithm then checks if an *interval* amount of time has passed since the last execution of Q (line 4). If an *interval* amount of time has passed, the algorithm re-executes query Q on model M. In practice, Q is defined as follows:

```
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT DISTINCT ?x
WHERE { ?x twitter:Has_Friend ?y }
```

This query retrieves all Twitter users currently stored in the system and stores this result in variable R. Then, for every node r in the result-set R, the algorithm simply emits a tuple containing the user-identifier of that node to all bolts in the topology (line 7). Note that this process is periodically repeated until the entire topology is terminated.

Analyze-Topology - Bolts

In this subsection, we present a detailed description of the various bolts that are implemented as a part of the Analyze-Topology. These bolts perform centrality estimation for the different Twitter users currently stored in the system.

The **Degree-C-Bolt** is used to compute the degree centrality for all nodes currently stored in the system. Algorithm 11 presents a sequence of steps used to perform this task.

Algorithm 11 DEGREE-C-BOLT()

Input: *tuple*, $totalNodes$, HBase-Nodes-View

1: $node \leftarrow tuple.getString(0)$
2: $degree \leftarrow 0$
3: $adjList \leftarrow$ HBase-Nodes-View.$get(node,$ Adj-List$)$
4: **for** $adjNode \in adjList$ **do**
5: $\quad degree \leftarrow degree + 1$
6: **end for**
7: $degC \leftarrow \frac{degree}{totalNodes-1}$
8: HBase-Nodes-View.$add(node,$ DegC$, degC)$

Algorithm 11 takes as input a tuple containing the user-identifier of a Twitter user. Additionally, the algorithm also takes as input, the total number of Twitter users in the system, $totalNodes$, and the Node-centric view, HBase-Nodes-View. The algorithm begins by extracting the string representation of the user-identifier, *node*, from the input tuple (line 1). In addition, the algorithm also initializes a variable, *degree* to 0 (line 2). This variable denotes the number of "friendship" links the current user has with other Twitter users. Then, the algorithm obtains the adjacency list, $adjList$, for the current user by using HBase-Nodes-View (line 3). For every node, $adjNode$, in this list the algorithm simply increments *degree* by 1 (line 5). Algorithm 11 then computes the degree centrality of the current node, $degC$, as a fraction of *degree* over $totalNodes - 1$ (line 7). The term $totalNodes - 1$ is used in the denominator since the "friendship" link is undirected. Finally, the algorithm updates the Node-centric view with the $degC$ value (line 8).

The **Closeness-C-Bolt** is used to compute the closeness centrality for all Twitter users currently stored in the system. Algorithm 12 presents a series of successive steps that can be used to perform this task.

Algorithm 12 receives as its input a tuple containing the user-identifier of a Twitter user whose closeness centrality we need to compute. In addition, the algorithm also receives the following parameters as inputs: (i) $totalNodes$ - the total number of Twitter users currently stored in the system. (ii) HBase-Nodes-View - the Node-centric view that stores metadata about the nodes in the Twitter network. (iii) HBase-Landmarks-View - the Landmark-centric view that stores path-related information for paths that connect non-landmark nodes with landmark nodes. The algorithm first extracts the user-identifier of the current Twitter user in the variable *node* (line 1) and also initializes a separate

Algorithm 12 CLOSENESS-C-BOLT()

Input: *tuple*, *totalNodes*, HBase-Nodes-View, HBase-Landmarks-View

1: $node \leftarrow tuple.getString(0)$
2: $sumOfPaths \leftarrow 0$
3: $closestLandmark \leftarrow$ HBase-Nodes-View.$get(node,$ Closest-Landmark)
4: **if** $node == closestLandmark$ **then**
5: $initDistance \leftarrow 0$
6: **else**
7: $initDistance \leftarrow$ HBase-Nodes-View.$get(node,$ Dist-To-Closest-Landmark)
8: **end if**
9: $landmarkAndNodes \leftarrow$ HBase-Landmarks-View.$get(closestLandmark)$
10: **for** $landmarkAndNode \in landmarkAndNodes$ **do**
11: $sumOfPaths \leftarrow sumOfPaths+$HBase-Landmarks-View.$get(landmarkAndNode,$ Distance)+ $initDistance$
12: **end for**
13: $closeC \leftarrow \frac{totalNodes-1}{sumOfPaths}$
14: HBase-Nodes-View.$add(node,$ CloseC, $closeC)$

variable $sumOfPaths$ to 0 (line 2). This variable denotes the sum of paths from *node* to all other nodes in the network. The algorithm then finds the landmark node that is closest to *node* (*closestLandmark*) by performing a look-up in HBase-Nodes-View (line 3). Next, the algorithm initializes a variable, *initDistance* to 0 if *node* is the same as *closestLandmark* else it is initialized with the distance between *node* and *closestLandmark*, which can be obtained from HBase-Nodes-View (lines 4-7). The algorithm then retrieves all rows containing *closestLandmark* from HBase-Landmarks-View (line 9). For each row in this set, the algorithm increments $sumOfPaths$ with the *initDistance* from *node* to *closestLandmark* and the distance from *closestLandmark* to other non-landmark nodes (lines 10-12). Recall that a row key in HBase-Landmarks-View is made up of a LandmarkId and a NodeId, where the NodeId represents some non-landmark node. Then, Algorithm 12 computes the closeness centrality of the current node, *closeC*, as a fraction of $totalNodes - 1$ over $sumOfPaths$ (line 13). Finally, the algorithm updates the Node-centric view with the *closeC* value (line 8).

The **Betweenness-C-Bolt** is used to compute the betweenness centrality for nodes belonging to the Twitter network that are currently stored in the system. Algorithm 13 presents a series of successive steps that can be used to perform this task.

Algorithm 13 receives the same input parameters that were received by Algorithm 12. The algorithm begins by extracting the user-identifier of the current Twitter user in the variable *nodeToCompute* (line 1) and also initializes a separate variable *betweenness* to 0 (line 2). This variable denotes the betweenness for the current Twitter user. Next, the algorithm retrieves all rows

from the HBase-Nodes-View (line 3). For each *node* in this set, the algorithm first retrieves the landmark node that is closest to *node* (*closestLandmark*) by performing a look-up in HBase-Nodes-View (line 5). This is followed by retrieving all rows from HBase-Landmarks-View that contain *closestLandmark* (line 6). For every row in this set, the algorithm first retrieves the number of paths (*numOfPaths*) to the non-landmark node (line 8), represented by the NodeId part of the row key of HBase-Landmarks-View. In addition, all actual paths (*listOfPaths*) are also retrieved (line 9). For every path (*path*) in this list, the algorithm first checks if *path* contains the current Twitter user (*nodeToCompute*). If it does, the algorithm increments the variable *numOfPathsWithNode* that keeps track of the number of paths that contain *nodeToCompute* (line 13). Once the algorithm iterates over all available paths to a non-landmark node, the algorithm updates *betweenness* with the fraction of *numOfPaths* that contain *nodeToCompute*, namely *numOfPathsWithNode* (line 16). Finally, after the algorithm iterates over all nodes, it normalizes the value of *betweenness* to compute the betweenness centrality, *betC* (line 19) which is then stored in HBase-Nodes-View (line 20). Again, the normalization factor is $\frac{(n-1)\times(n-2)}{2}$ since the network is undirected.

Algorithm 13 BETWEENNESS-C-BOLT()

Input: *tuple*, *totalNodes*, HBase-Nodes-View, HBase-Landmarks-View

1: $nodeToCompute \leftarrow tuple.getString(0)$
2: $betweenness \leftarrow 0$
3: $allNodes \leftarrow$ HBase-Nodes-View.$getRowKeys()$
4: **for** $node \in allNodes$ **do**
5: $closestLandmark \leftarrow$ HBase-Nodes-View.$get(node,$ Closest-Landmark$)$
6: $landmarkAndNodes \leftarrow$ HBase-Landmarks-View.$get(closestLandmark)$
7: **for** $landmarkAndNode \in landmarkAndNodes$ **do**
8: $numOfPaths \leftarrow$ HBase-Landmarks-View.$get(landmarkAndNode,$ Num-Of-Paths$)$
9: $listOfPaths \leftarrow$ HBase-Landmarks-View.$get(landmarkAndNode,$ Paths$)$
10: $numOfPathsWithNode \leftarrow 0$
11: **for** $path \in listOfPaths$ **do**
12: **if** $path.contains(nodeToCompute)$ **then**
13: $numOfPathsWithNode \leftarrow numOfPathsWithNode + 1$
14: **end if**
15: **end for**
16: $betweenness \leftarrow betweenness + \frac{numOfPathsWithNode}{numOfPaths}$
17: **end for**
18: **end for**
19: $betC \leftarrow \frac{betweenness\times 2}{(n-1)\times(n-2)}$
20: HBase-Nodes-View.$add(node,$ BetC$, betC)$

Chapter 4

Performance Evaluation

This chapter presents the details of our preliminary experimental investigation into the performance of StormRider. We begin by presenting details of the experimental setup used to conduct our experiments. We also give a brief description of the dataset that we have used in our experiments. We then present our findings that detail the effectiveness of the various topologies implemented in StormRider.

4.1 Experimental Setup

We conducted our experimental evaluation on a local cluster containing 14 nodes. Each machine in this cluster consists of a Pentium IV processor with 290GB to 360GB disk space and 4GB main memory. This cluster ran Hadoop v0.20.2 and HBase v0.90.1. The nodes are connected with a 48-port Cisco switch on an internally created private network.

Since we compared our landmark-based approximation technique for centrality estimation with the exact method from [17], we also briefly describe the configuration of the machine on which the exact method was evaluated. The experiments for exact centrality computation were conducted on a machine that used an Intel Core2 Duo processor with a 250GB hard drive and 4GB main memory. The experiments used JRE v1.6.0_22 as the Java engine.

4.2 Experimental Dataset - Twitter

To demonstrate the efficacy of StormRider, we used the Twitter network as the basis for the implementation of the various Add-, Query- and Analyze-Topologies. Therefore, we also made use of the Twitter dataset to conduct our preliminary experimental investigation into StormRider. In particular, we gathered information (profile and tweet) about 500,000 Twitter users, who were randomly selected, using the Twitter Rest API [18]. Note that the information for each Twitter user was gathered twice, where the interval between the

two instances of information collection was a few months (between 3-6), thus constructing two separate snapshots of the Twitter network. The selected interval of time allowed us to see clear changes in the network, particularly in the neighborhood of some of the more active Twitter users.

A Twitter user profile consists of basic information such as name, location, bio description, count of followers/friends, the latest tweet etc. A user's tweet contains information such as when the tweet was created, the message of the tweet, it's source if any, etc. For our preliminary experiments, we used our Add-Topology to add this information to StormRider. Simultaneously, we executed the Analyze-Topology to compute centrality metrics for each of these users.

4.3 Experimental Evaluation

In this section, we present details of the experimental evaluation that we have carried out for StormRider. We begin by giving a brief description of the procedure used in our experimental evaluation. This is followed by a detailed comparison of our landmark-based approximation method used to compute closeness and betweenness centrality with exact methods that are also used to perform the same task. We have not conducted an evaluation for degree centrality since no approximation technique is needed to compute it, we can always compute an exact value using a node's adjacency list and the total number of nodes in the network at any given time. Also, we have neither added nor evaluated other social network analysis (SNA) metrics, since we first wanted to build up a prototype that supports automated addition, retrieval and basic analysis of a network. In the future, we plan to expand our current work with additional SNA metrics. Finally, we have not presented any experimental evaluation related to SPARQL queries, since the performance of these queries is dependent on Jena-HBase. The performance of different types of SPARQL queries on Jena-HBase was extensively studied in [7].

4.3.1 Experimental Procedure

As stated earlier, we evaluated our centrality estimation experiments on a maximum of 500,000 users of the Twitter network. In this evaluation, we used the landmark-based approximation technique [10, 11] to compute shortest paths between any two nodes in the network. For our experiments, the number of landmark nodes was selected as, the total number of nodes in the graph / 100. Additionally, the Degree strategy given in [10] was used to select the elements in the set of landmark nodes. Note that the number of landmarks was randomly selected but as is shown in Figure 3 of reference [10], at this order of magnitude (*viz* 10^3), the Degree technique produces a very small approximation error (*approx* 0.03-0.09). Finally, recall that in our current work we only focus on the "friendship" link between Twitter users which means that our experiments are evaluated on an undirected graph.

4.3.2 Performance of StormRider for Closeness Centrality

Aim: The aim of this experiment was to evaluate the efficiency of the HBase views in being able to compute the closeness centrality metric. Thus, the evaluation is in turn dependent on the number of landmarks selected as well as the mechanisms employed to update the views by the Add-Topology.

Procedure: In this experiment, we computed the closeness centrality for increasing graph sizes (from 100,000 to 500,000 nodes). The closeness centrality was computed by using the shortest paths obtained through the landmark-based approximation technique in our implementation, and the exact technique given in reference [17]. Note that since we constructed two different snapshots of the Twitter network, we automatically computed the closeness centrality for each snapshot in turn using the topologies described in an earlier chapter. Then, we analyzed the performance of both methods across the following metrics:

1. Approximation Error: The approximation error measures the accuracy of the approximate method (based on a landmark-based technique for shortest path estimation) in computing the closeness centrality. The approximation error was computed as follows: $|\hat{l} - l|/l$ where l is the actual closeness centrality and $\hat{l}$ is the approximation.

2. Overall Execution Time: The overall execution time measures the time required to perform the approximate and exact computation of closeness centrality. The overall execution time for the approximate case is computed as a sum of both, the time to update the views when nodes are re-discovered (Node-centric and Landmark-centric View blocks in Figure 3.4) and the time to perform the actual closeness centrality computation (Closeness-C-Bolt).

3. User-centric Metric Time: This metric measures the time required to compute closeness centrality for a single user using the landmark-based technique across both snapshots of the Twitter network. In practice, the user-centric metric time is estimated as the time required by the Closeness-C-Bolt to compute closeness centrality for the selected user. The user was randomly selected and is a non-landmark node. In addition, the number of friends in the selected user's neighborhood changed from 175 to 202 across the two Twitter snapshots.

Note that the graphs for Approximation Error and Overall Execution Time given below present results of closeness centrality computation over the second Twitter snapshot.

Observations: We now present our observations for each of the experimental metrics we outlined above.

1. Approximate Error: The leftmost graph of Figure 4.1 presents the results of the approximation error experiment for StormRider. We clearly see that

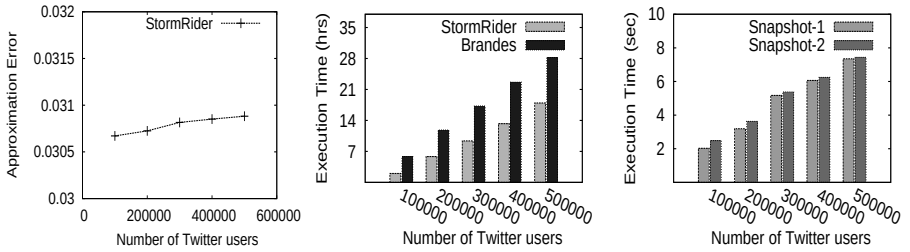

Figure 4.1: Performance of StormRider for Closeness Centrality computation

the error remains small even when the number of Twitter users increases from 100K to 500K. The magnitude of the error is small as is expected since when the number of landmark nodes is on the order of 10^3, the approximation error is small as shown in reference [10].

2. Overall Execution Time: The graph in the center of Figure 4.1 presents the results of the experiment that measures the overall execution time of StormRider's approximate algorithm *vs.* Brandes's exact algorithm given in [17]. We observe that the execution time of StormRider's approximate algorithm is much better than Brandes's algorithm. This is expected, since the exact algorithm requires k single-source-shortest-path (SSSP) computations while the approximate algorithm requires $k/100$ SSSP computations and k BFS executions which leads to a lesser overall execution time (where k is the number of nodes in the network).

3. User-Centric Metric Time: The rightmost graph of Fig. 4.1 presents the results of the experiment that measures the time to compute closeness centrality for the preselected user across both snapshots of the Twitter network. Firstly, we observe that the difference between execution times across both snapshots is very small for the entire range (100K-500K). This is because the two views used by the Closeness-C-Bolt during centrality estimation for the selected user are nearly identical across both snapshots. We also observe that the execution time consistently increases across the entire range (100K-500K). This slight increase is due to a longer lookup time in the Landmark-centric view, which increases in size as the number of users is increased from 100K to 500K.

4.3.3 Performance of StormRider for Betweenness Centrality

Aim: The goal of this experiment was to evaluate the efficiency of the HBase views in being able to compute the betweenness centrality metric. As before, the evaluation depends on the number of landmarks selected as well as the implementation of the different bolts in the Add-Topology that are used to update

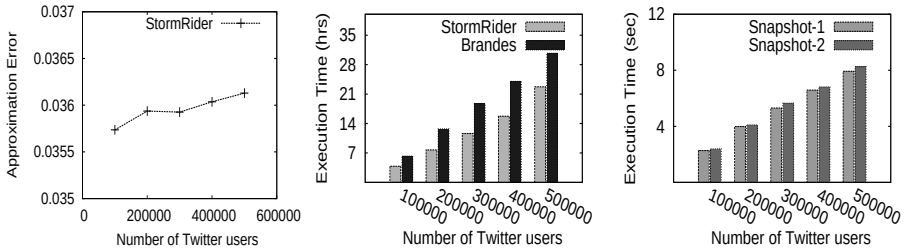

Figure 4.2: Performance of StormRider for Betweenness Centrality computation

the views.

Procedure: The procedure for this experiment was the same as the one for the previous experiment that computes closeness centrality, the only difference being that in this case we compute betweenness centrality in all cases.

Observations: We now present our observations for each of the experimental metrics that were outlined above. We also notice that the results in Figure 4.2 are very similar to the results given in Figure 4.1. This is because both, closeness and betweenness centrality make use of shortest paths in their computation. Furthermore, the algorithms that compute shortest paths remain relatively the same in both of sets experiments, thereby leading to nearly identical results.

1. Approximate Error: The leftmost graph of Figure 4.2 presents the results of the approximation error experiment for betweenness centrality estimation in StormRider. As before, we clearly see that the error remains small even when the number of Twitter users increases from 100K to 500K. The magnitude of the error is small as is expected since when the number of landmark nodes is on the order of 10^3, the approximation error is small similar to the results given in reference [10].

2. Overall Execution Time: The graph in the center of Figure 4.2 presents the results of the experiment that measures the execution time of StormRider's approximate algorithm *vs.* Brandes's exact algorithm given in [17] for betweenness centrality computation. We observe that the execution time of StormRider's approximate algorithm is much better than Brandes's algorithm. This is expected, since the exact algorithm requires k all-pairs-all-shortest-path (APASP) computations while the approximate algorithm requires $k/100$ SSSP computations, k BFS executions and k^2 iterations over all nodes in the network, which leads to a lesser overall execution time (where k is the number of nodes in the network). Additionally, the execution time for both cases is slightly greater than their corresponding counterparts in Figure 4.1. This is because the SSSP computations in the previous case are now replaced by APASP calculations or multiple iterations over the nodes in the network.

3. User-Centric Metric Time: The rightmost graph of Fig. 4.2 presents the results of the experiment that measures the time to compute betweenness centrality for the preselected user across both snapshots of the Twitter network. Again, we observe that the difference between execution times across both snapshots is very small for the entire range (100K-500K). This is because the views used by the Betweenness-C-Bolt are nearly identical. As before, we also see that the execution time consistently increases across the entire range (100K-500K). This is due to a longer lookup time in the Landmark-centric view, which increases in size as the number of users increases from 100K to 500K.

Chapter 5

Related Work

In this chapter, we present a summary of the existing work done in the areas of storage, query and analysis of social networks in the context of Cloud Computing and Semantic Web technologies. In addition, we also present a brief survey of existing approximation techniques for centrality estimation, since we use one such technique (landmarks-based) in our work.

Graph Databases: A graph database uses the concept of a graph (*viz.* a set of nodes connected by edges) to represent and store data. In addition, graph databases are optimized for answering graph-like queries. Since a number of graph databases have been developed by the research community, in the subsequent discussion we limit our attention to graph databases that use an RDF representation to store data. A point to note is that although the frameworks described in the discussion below use RDF as its data model for representing networks, they either suffer from scalability issues (*e.g.* AllegroGraph) or are incapable of querying or analyzing networks as they evolve (*e.g.* Bigdata and Virtuoso). These drawbacks do not allow the described frameworks to handle interesting use cases that arise due to the evolving nature of networks.

AllegroGraph is a database and application framework for building Semantic Web applications [19]. It allows storing data as RDF triples as well as querying these triples through various query APIs like SPARQL and Prolog. Additionally, AllegroGraph provides RDFS++ reasoning capabilities with its built-in reasoner. AllegroGraph also includes support for Federation, Social Network Analysis, Geospatial capabilities and Temporal reasoning. The AllegroGraph SNA tool provides a library that allows us to treat a RDF triple store as a social network with the ability to calculate centrality, importance, as well as different search functions. Examples of centrality for actors and groups are degree, betweenness and closeness. Additional capabilities provided with the SNA tool are techniques for finding cliques and shortest paths between actors. The advantage of using this tool is that the representation of the social network is in RDF while the drawback is scalability, since this tool works on a single workstation.

Bigdata provides a high performance RDF database that is capable of providing a fast load throughput and best-in-class query performance [20]. Addi-

tionally, the Bigdata RDF database allows users to perform large-scale semantic alignment as well as data set federation. Further, the Bigdata RDF database provides RDFS and limited OWL inference as well as full text search and indexing using Lucene. Finally, the Bigdata RDF database also allows users to define custom rules for conducting inference as well as performing analysis on networks stored in the database.

The Virtuoso Universal Server [21] uses a hybrid server architecture to support the functionalities of various data models such as the relational model, RDF, XML, *etc.* The Virtuoso RDF Quad-Store [22] provides the basic capabilities required of any RDF framework such as storage in several formats (HTML+RDFa, RDF-JSON, N3, Turtle, TriG, TriX, and RDF/XML), SPARQL query processing and inference (using a backward chaining OWL reasoner). In addition, the store also provides security features, indexing capabilities and SPARQL extensions for geo-spatial queries, full-text queries, *etc.*

Social Network Analysis tools: The rapid growth of web and online social graphs in the past decade has resulted in the development of several frameworks that provide efficient strategies for analyzing these graphs. In the discussion below, we provide a brief summary of a few of these frameworks along with some of their key features. A point to note is that although the frameworks described below provide analysis and mining capabilities on large networks, they are unable to automatically perform these tasks as a network evolves over time. To perform this task, these frameworks require a user to manually enter the different snapshots of a network, which are then analyzed to detect changes across them.

General purpose social network analysis tools: JUNG [23] is a software library for modeling, analyzing and visualizing data that can be represented as a graph. Towards this end, JUNG provides support for various graph representations such as directed and undirected graphs, hypergraphs, *etc.* Additionally, JUNG includes implementations of various graph algorithms such as clustering, importance measures, statistical analysis, *etc.* Finally, JUNG also provides tools that allow users to explore a graph using a built-in or custom designed visualization layout.

*ORA is a dynamic meta-network assessment and analysis tool developed by CASOS at Carnegie Mellon [6]. It contains metrics for analyzing static and dynamic networks as well as techniques for comparing networks at various levels (network-level, group-level and individual-level). Additionally, *ORA can also be used to examine how networks change over time through the analysis of trail data; in this respect, *ORA is similar to StormRider, which tries to detect changes that occur in an evolving network. However, to perform this task, *ORA requires a user to manually enter the different snapshots of a network, which are then analyzed to detect changes across them.

The work given in [2] provides formal definitions of SNA operators in SPARQL. In addition, the authors also propose SemSNA, an ontology that models SNA characteristics, which can then be used to annotate social networks. However, rather than model SNA operators in SPARQL, we have created topologies for them in StormRider. This allows us to perform analysis tasks on larger graphs

than those given in [2] while maintaining a reasonable level of performance.

Cloud-based social network analysis tools: There are several cloud-based tools that provide social network analysis capabilities. For example, X-RIME [5] provides a Hadoop MapReduce based library that allows us to store an adjacency list representation of a social network with different analysis functions. Examples of functions provided are strongly and weakly connected components, shortest paths, maximal cliques, pagerank, vertex degree statistics *etc.* The advantage of this tool is its scalability due to the use of Hadoop.

PEGASUS is a Peta-scale graph mining system [3] that uses the Hadoop MapReduce framework to provide functions for computing degree distribution and pagerank, radius estimation, connected components, triangle counting and Random Walks with Restart (RWR). Again, the advantage of PEGASUS is its scalability since it uses the Hadoop MapReduce framework. However, as with X-RIME, PEGASUS only provides analytical capabilities for a network while ignoring aspects related with the storage and retrieval of networks.

HaLoop [24] is a modified version of Hadoop that provides built-in support for iterative processing, which is an implicit part of graph processing. HaLoop achieves this goal by using various caching mechanisms that make Hadoop's task scheduler loop-aware. Additional features of HaLoop include: (i) Allowing users to reuse building blocks from standard Hadoop iterative application implementations. (ii) Providing intra-job fault tolerance similar to Hadoop.

Pregel [25] and Apache Giraph [26] provide cloud-based computational models for efficient processing of large graphs. A processing task in these tools is implemented as a sequence of iterations, where in an iteration, a node can receive messages from a previous iteration and can itself send messages to other nodes. Additionally, a node can also modify its own state as well as that of its outgoing edges. Furthermore, this node-centric approach is general enough to support a number of analysis algorithms. Finally, since these tools are cloud-based they provide several advantages such as efficiency, scalability, reliability and fault tolerance. The key difference between Pregel and Giraph is that the latter adds fault tolerance to the processes that control the coordination between multiple iterations of a task by making use of ZooKeeper as its centralized coordination service.

SNAP is a C++ framework for network analysis and graph mining [27]. SNAP can be used to generate and manipulate large graphs (regular and random), computing structural properties of a network such as the number of triads, and supporting attributes for nodes and edges in a network.

GraphLab is a framework that compactly expresses asynchronous iterative machine learning (ML) algorithms that have sparse computational dependencies while ensuring data consistency and achieving a high degree of parallel performance [28]. GraphLab provides the following analog to the Map and Reduce abstractions: (i) Map $\rightarrow$ Update Function: This function allows reading and modifying overlapping sets of data. Additionally, this function can be triggered recursively, thus allowing a chain of update functions to be applied to a graph. (ii) Reduce $\rightarrow$ Sync Operation: This function allows a reduce operation to be performed in the background, even when other computations are being per-

formed. Also, this operation allows reading multiple records at the same time, thus enabling a task to execute on larger dependent contexts.

Cassovary [29] is a "big graph" processing library that is designed to handle graphs with billions of nodes and edges in a space-efficient manner. Cassovary provides the basic tools and algorithms necessary to conduct the analysis and mining of large networks.

Approximation algorithms for centrality metrics: Several approaches for approximating centrality metrics, particularly closeness and betweenness, have been proposed by the research community. These include sampling based techniques [30, 31, 32]. The idea behind this technique is to repeatedly select nodes from a network based on a sampling technique (random, adaptive, *etc.*). Then, the required shortest path computations are performed using each of the sampled nodes as the starting point. The results of these calculations are then used to estimate centrality values for the remaining nodes in the network. A parallel version of the algorithm given in [30] is implemented in [33].

In [17], Brandes presents an exact calculation of betweenness centrality. The algorithm has the following steps: (i) Solve the single-source-shortest-path (SSSP) problem for each node of the network. (ii) After each iteration, the dependencies of the source (s) on each other node (v) are added to the centrality score of that node, where a dependency of s on v is the fraction of all paths starting from s on which v lies. A parallel version of this exact algorithm is implemented in [33]. An approximate version of this algorithm is constructed in [34] by extrapolating from a subset of k *pivots* but still using the aggregation strategy of the exact algorithm. In [35], the authors extend the work of [34] to improve the betweenness for nodes that are overestimated as a result of being near nodes that are selected as pivots. They achieve their goal by using an unbiased estimator and a scaling function.

The authors in [10] study approximate-*landmark-based* methods for path estimation in large networks. Their approach involves selecting a set of landmarks and computing the shortest distance from all other nodes to these nodes offline. Then, at runtime when the shortest distance for a pair of nodes is needed, it is estimated using these precomputed distances. Additionally, in reference [10], the authors prove that the landmark selection problem is NP-Hard and they subsequently present different landmark-selection strategies such as Random, Degree and Centrality, along with constrained and partitioned variants of them.

In [11], the authors introduce the concept of a network structure index (NSI) that consists of a set of annotations for every node in the network, and a function that uses the annotations to estimate graph distances between pairs of nodes. Formally, for a graph $G(V, E)$, annotations define a function $A : V \rightarrow S$, where S is an arbitrarily complex annotation space, and the other element of the index is a distance measure $D : S \times S \rightarrow \Re$ that maps pairs of node annotations to a positive real number. The NSI's are based on different measures such as degree, landmark-based, *etc.* To estimate closeness centrality, they calculate the average of the graph distance to a sample of nodes in the data set. They use the NSI distance estimate to calculate the distance between these sampled pairs. To compute betweenness centrality, they randomly sample pairs of nodes

and discover the shortest path between them and then count the number of times a given node appears on this path. They contend that with this method they can find somewhat accurate betweenness centralities with a much smaller proportion of nodes visited than with traditional breadth-first search.

The authors in [36] present a set of parallel algorithms to compute closeness and ego-centric betweenness centrality. They define an anytime anywhere methodology for this purpose. Anytime refers to providing results at any given time and refining these results over time. Anywhere refers to incorporating new information into the whole network. Their methodology consists of three parts: (i) Domain Decomposition: In this phase the graph is partitioned into subgraphs such that the cut size is minimized. (ii) Initial Approximation: Any SNA tool is used to analyze each partition on a separate processor. (iii) Recombination: This phase incrementally combines the partial results to obtain the final results.

Chapter 6

Conclusions and Future Work

There has been a significant body of research devoted to the storage, retrieval and analysis of online social networks using cloud computing and semantic web technologies in conjunction with existing social network analysis measures. However, all of this work considers a network as a series of snapshots. Therefore, all of the network operations (*viz.* storage, retrieval and analysis) need to be independently performed on each of these snapshots. In reality, online social networks continuously evolve and as such we need to have in place a mechanism that allows us to perform network operations as the network is changing.

In this report, we have presented StormRider, that aims to address this issue by using a novel combination of existing cloud computing and semantic web technologies. StormRider provides an application programmer with the necessary tools for developing and executing various operations on networks of their choice. Additionally, the ability to interact with evolving networks opens up several new, realistic use cases that can now be supported. For example, tracking the neighborhood of a given node in the network and being able to store and access previous states of a network. In this report, we presented a prototype of StormRider that mainly interacts with the Twitter network. We also gave a preliminary experimental evaluation of this prototype. We further plan to explore the following areas of research with regard to StormRider:

- In the current iteration of our work, we have built a prototype that uses Twitter as its underlying network of choice. In the future, we plan to build topologies that interact with various other networks such as Google Plus, Foursquare, *etc.*
- We also plan to implement algorithms for various other social network analysis metrics such as diameter estimation, models of information flow, identification of nodes that act as influencers *etc.*
- In our current work we have only focused on storing information related to

a single network (*viz.* Twitter) in a RDF representation in an underlying Jena-HBase model. However, one of the main design goals of the Semantic Web is to allow integration of data from multiple sources. Therefore, in the future we plan to investigate how data from multiple networks can be effectively integrated into the same model. This direction of research is quite challenging since it requires the design of ontologies that allow data integration in the presence of ambiguities in data.

Bibliography

[1] World Wide Web. http://en.wikipedia.org/wiki/World_Wide_Web.

[2] G. Erétéo, M. Buffa, F. Gandon, and O. Corby. Analysis of a Real Online Social Network Using Semantic Web Frameworks. In A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, editors, *International Semantic Web Conference*, volume 5823 of *Lecture Notes in Computer Science*, pages 180–195. Springer, 2009.

[3] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System. In W. Wang, H. Kargupta, S. Ranka, P. S. Yu, and X. Wu, editors, *ICDM*, pages 229–238. IEEE Computer Society, 2009.

[4] PEGASUS: Peta-Scale Graph Mining System. http://www.cs.cmu.edu/~pegasus/.

[5] X-RIME: Hadoop based large scale social network analysis. http://xrime.sourceforge.net/.

[6] ORA. http://www.casos.cs.cmu.edu/projects/ora/.

[7] V. Khadilkar, M. Kantarcioglu, P. Castagna, and B. Thuraisingham. Jena-HBase: A Distributed, Scalable and Efficient RDF Triple Store. Technical report, 2012. http://www.utdallas.edu/~vvk072000/Research/Jena-HBase-Ext/tech-report.pdf.

[8] S. Ghemawat, H. Gobioff, and S-T Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[10] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876, 2009.

[11] M. J. Rattigan, M. Maier, and D. Jensen. Using structure indices for efficient approximation of network properties. In T. Eliassi-Rad, L. H.

Ungar, M. Craven, and D. Gunopulos, editors, *KDD*, pages 357–366. ACM, 2006.

[12] L.T. Detwiler, D. Suciu, and J.F. Brinkley. Regular paths in SparQL: querying the NCI thesaurus. In *AMIA Annual Symposium Proceedings*, volume 2008, page 161. American Medical Informatics Association, 2008.

[13] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: transaction time support for SQL server. In F. Özcan, editor, *SIGMOD Conference*, pages 939–941. ACM, 2005.

[14] Social Media Application. `https://www.fbo.gov/index?s=opportunity&mode=form&id=c65777356334dab8685984fa74bfd636&tab=core&tabmode=list&=`.

[15] Open Source Indicators (OSI) Program Broad Agency Announcement. `https://www.fbo.gov/index?s=opportunity&mode=form&id=cf2e4528d4cbe25b31855a3aa3e1e7c9&tab=core&_cview=0`.

[16] Social Media in Strategic Communication (SMISC). `https://www.fbo.gov/index?_cview=0&id=6ef12558b44258382452fcf02942396a&mode=form&s=opportunity&tab=core`.

[17] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[18] Twitter REST API. `https://dev.twitter.com/docs/api`.

[19] AllegroGraph SNA tool. `http://www.franz.com/agraph/support/documentation/current/agraph-introduction.html`.

[20] Bigdata RDF Database. `http://www.systap.com/bigdata.htm`.

[21] Virtuoso Universal Server. `http://virtuoso.openlinksw.com/`.

[22] Virtuoso RDF Quad-Store. `http://virtuoso.openlinksw.com/rdf-quad-store/`.

[23] Java Universal Network/Graph Framework (JUNG). `http://jung.sourceforge.net/`.

[24] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1):285–296, 2010.

[25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD Conference*, pages 135–146. ACM, 2010.

[26] Apache Giraph. `http://incubator.apache.org/giraph/index.html`.

[27] Stanford Network Analysis Package (SNAP). `http://snap.stanford.edu/snap/index.html`.

[28] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[29] Cassovary. `https://github.com/twitter/cassovary`.

[30] D. Eppstein and J. Wang. Fast Approximation of Centrality. *J. Graph Algorithms Appl.*, 8:39–45, 2004.

[31] K. Okamoto, W. Chen, and X-Y Li. Ranking of Closeness Centrality for Large-Scale Social Networks. In *FAW*, pages 186–195, 2008.

[32] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating Betweenness Centrality. In *WAW*, pages 124–137, 2007.

[33] D. A. Bader and K. Madduri. Parallel Algorithms for Evaluating Centrality Indices in Real-World Networks. In *ICPP*, pages 539–550. IEEE Computer Society, 2006.

[34] U. Brandes and C. Pich. Centrality Estimation in Large Networks. *I. J. Bifurcation and Chaos*, 17(7):2303–2318, 2007.

[35] R. Geisberger, P. Sanders, and D. Schultes. Better Approximation of Betweenness Centrality. In *ALENEX*, pages 90–100, 2008.

[36] E.E. Santos, P. Long, D. Arendt, and M. Pittkin. An Effective Anytime Anywhere Parallel Approach for Centrality Measurements in Social Network Analysis. In *SMC*, volume 6, pages 4693–4698, October 2006.